UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

USO DE ANÁLISIS ASOCIATIVO EN ALGORITMOS DE APRENDIZAJE

Autor: Gonzalo Castro Casillas
Tutor: Estrella Pulido Cañabate

AGRADECIMIENTOS

Gracias a toda la gente que me ha apoyado estos años duros de carrera, gracias a mis compañeros por hacerme las clases y el estudio entretenido, sobre todo gracias a Fide, a Jhon y a Carlangas.

Gracias a Estrella por su dedicación, compromiso e inestimable ayuda.

Y por supuesto gracias a mi familia sin la que no podría haber llegado hasta este punto.

RESUMEN

El estudio que hemos realizado para este trabajo de fin de grado se enmarca en el campo de la Minería de Datos y el Aprendizaje Automático. Más concretamente este proyecto se encuadra en el análisis de algoritmos de aprendizaje y en su implementación en el entorno Hadoop, un sistema de código abierto que permite procesar grandes volúmenes de datos de forma distribuida.

El análisis asociativo es un método de aprendizaje automático que se utiliza para descubrir relaciones ocultas en grandes conjuntos de datos. Una forma eficiente de aplicar el análisis asociativo es utilizando el algoritmo "A priori".

El objetivo de este trabajo consiste en analizar la ejecución del algoritmo "A priori" tanto de forma secuencial como aplicando el modelo de programación MapReduce, para comprobar cuáles son las mejoras de rendimiento cuando éste es ejecutado en paralelo.

Para poder llevar a cabo este trabajo, primero se realizó un estudio exhaustivo del algoritmo "A priori", para comprender su funcionamiento de forma matemática. A continuación, se implementó dicho algoritmo en código java y se llevaron a cabo pruebas de forma secuencial para comprobar su correcto funcionamiento, midiendo los tiempos de ejecución.

Una vez implementado el código y ejecutado el algoritmo, hubo que realizar la transformación del mismo al paradigma Hadoop MapReduce implementando las funciones Map y Reduce.

Para finalizar, se realizaron una batería de pruebas para comprobar que los tiempos en paralelo van mejorando una vez que incrementamos el número de nodos, con respecto a los tiempos de forma secuencial.

Las pruebas realizadas demostraron que la propuesta era acertada ya que se observó una reducción sustancial de los tiempos de procesamiento.

Palabras Clave: A priori, Hadoop, MapReduce, Nodos, Aprendizaje Automático, Minería de Datos.

SUMMARY

The study we have conducted for this final degree work is framed into the fields of Data Mining and Machine Learning. Specifically this project is related to the learning algorithms analysis and their implementation in the Hadoop framework, an open-source system that allows big data sets to be processed in a distributed mode.

Associative analysis is a machine learning method used to discover hidden relations within big data sets. An efficient way of applying associative analysis is by using the "A priori" algorithm.

The final goal of this work is to analyze the execution of the "A priori" algorithm both in a sequential form, and by using the Hadoop technology, in order to test performance improvements when the algorithm is executed in parallel.

The first step was to perform an exhaustive study of the "A priori" algorithm, in order to understand its mathematic behavior. Afterwards, we proceeded to its implementation in java code form and a series of sequential form test were conducted to check its correct behavior, measuring its execution times.

Once the code has been implemented and executed, the transforming of the project into the Hadoop MapReduce form took place by implementing the Map and Reduce functions.

Finally, a series of tests were performed to check that when we increase the number of nodes, the processing times experience a significant improvement when compared to the sequential form of them.

The tests shown that the premise object of this study, was correct since a meaningful reduction of processing times was observed.

Keywords: A priori, Hadoop, MapReduce, Nodes, Machine Learning,
Data Mining.

INDICE

LISTA DE FIGURAS

LISTA DE TABLAS

CAPITULO 1
INTRODUCCION

En las ciencias de la computación, el aprendizaje automático es una rama de la inteligencia artificial cuyo objetivo es desarrollar técnicas que permitan a los ordenadores *aprender.* El objetivo del aprendizaje es crear programas capaces de generalizar comportamientos a partir de una información no estructurada suministrada en forma de ejemplos. Es por lo tanto un proceso de conocimiento inducido.

En muchas ocasiones este campo de aprendizaje automático tiene grandes similitudes con el de la estadística, ya que las dos disciplinas se usan para el análisis de grandes volúmenes de datos.

El aprendizaje automático tiene una amplia gama de aplicaciones, incluyendo motores de búsqueda, diagnósticos médicos, detección de fraude en el uso de tarjetas de crédito, análisis del mercado de valores, clasificación de secuencias de ADN, reconocimiento del habla y del lenguaje escrito, juegos y robótica.

La generación de grandes volúmenes de datos que está ocurriendo en los últimos años ha hecho que adquiera una gran importancia la aplicación del aprendizaje automático al Big Data.

El Big Data se define como el conjunto de herramientas informáticas destinadas al manejo, gestión y análisis de grandes volúmenes de datos de todo tipo, los cuales no pueden ser gestionados por las herramientas informáticas tradicionales.

El objetivo fundamental del Big Data es dotar de una infraestructura tecnológica a las empresas y organizaciones con la finalidad de poder almacenar, tratar y analizar de manera económica, rápida y flexible la gran cantidad de datos que se generan diariamente. Para ello es necesario el desarrollo y la implantación tanto de hardware como de software específicos que gestionen esta explosión de datos con el objetivo de extraer valor para obtener información útil para nuestros objetivos o negocios.

Una de las tecnologías fundamentales asociadas al Big Data es el modelo de programación MapReduce inventado por Google para procesar grandes volúmenes de datos en paralelo.

El Big Data tiene numerosas aplicaciones en la vida real. Expondremos a continuación tres ejemplos de uso de Big Data para la vida real.

Comprender y analizar a los clientes

Esta es una de las áreas de uso de Big Data más grandes y más divulgados hoy en día. Aquí el Big Data se utiliza para comprender mejor a los clientes y sus comportamientos y preferencias. Las empresas están interesadas en ampliar sus conjuntos de datos tradicionales con datos de los medios sociales. El navegador registra las búsquedas y los comentarios para obtener una imagen más completa de sus clientes. El gran objetivo, en muchos casos, es crear modelos predictivos que permitan tener conocimiento de que son los artículos que más desea el cliente y ofrecérselos.

Comprender y optimizar procesos de negocio

El Big Data también se utiliza para optimizar los procesos de negocio. Los minoristas pueden optimizar sus acciones sobre la base de las predicciones generadas a partir de los datos de medios sociales, las tendencias de búsqueda web y las previsiones meteorológicas. Uno de los procesos de negocio en particular que está viendo una gran cantidad de análisis Big Data es la cadena de suministro o la optimización de rutas de entrega.

El yo cuantificado

El Big Data no es sólo para las empresas y gobiernos, sino también para todos nosotros individualmente. Ahora nos podemos beneficiar de los datos generados a partir de dispositivos portátiles, tales como relojes o pulseras inteligentes. Para comprender mejor lo que estamos diciendo veamos el siguiente ejemplo: un brazalete recoge datos sobre el consumo de calorías, niveles de actividad, y nuestros patrones de sueño. A pesar de que cada individuo tiene acceso a todos estos datos, el valor real está en el análisis de los datos colectivos.

El análisis asociativo es un método de aprendizaje automático que se utiliza para descubrir relaciones ocultas en grandes conjuntos de datos. Una forma eficiente de aplicar el análisis asociativo es utilizando el algoritmo “A priori”. En este trabajo se implementa este algoritmo de forma secuencial y aplicando el modelo de programación MapReduce. La eficiencia de la paralelización del algoritmo se evalúa ejecutando dicha implementación en un clúster con uno, dos y cuatro nodos.

Además de esta introducción, el proyecto incluye otros cinco capítulos. En el **capítulo 2** explicaremos el algoritmo” A Priori” así como las implementaciones realizadas.

El **capítulo 3** trata sobre el modelo de programación MapReduce, en qué consiste y cómo funciona.

En **el capítulo 4** describiremos los algoritmos usados, cómo se han modificado, y cómo se ha implementado la tecnología de MapReduce.

En **el capítulo 5** analizaremos las pruebas que realizamos con los algoritmos en modo secuencial y paralelo.

Por último en el **capítulo 6** se presentan las conclusiones extraídas y se describen las líneas de trabajo futuro.

CAPITULO 2
ANALISIS ASOCIATIVO: CONCEPTOS BASICOS Y ALGORITMOS

2.1 Análisis Asociativo

Antes de entrar en la explicación del análisis asociativo que hemos usado para nuestra implementación vamos a poner un ejemplo de cómo y para qué se usa.

Muchas empresas recogen gran cantidad de datos en sus operaciones del día a día. Un ejemplo claro es una empresa de alimentación. Por ejemplo, cada vez que un cliente realiza una compra se guardan los datos de los artículos que ha comprado. La Tabla 1. Base de datos con 5 transacciones muestra un ejemplo que contiene cinco registros o transacciones. Cada transacción tiene un identificador (*TID*) y el conjunto de elementos o *ítems* que han sido comprados.

TID	ELEMENTO
1	{PAN, LECHE}
2	{PAN, PAÑALES, CERVEZA, HUEVOS}
3	{LECHE, PAÑALES, CERVEZA, COLA}
4	{PAN. LECHE, PAÑALES, CERVEZA}
5	{PAN, LECHE, PAÑALES, COLA}

Tabla 1. Base de datos con 5 transacciones

El análisis asociativo es la metodología que nos permite descubrir relaciones entre atributos en conjuntos de datos. Estas relaciones se denominan reglas de asociación. Por ejemplo de Tabla 1 podemos obtener la siguiente regla: {Pañales} → {Cerveza}. Esta regla sugiere que existe una relación entre la venta de pañales y cerveza ya que en general cuando la gente compra pañales también compra cerveza. La empresa de alimentación puede usar estos datos para identificar nuevas oportunidades de negocio. Un ejemplo claro sería poner los pañales en el stand de al lado de las cervezas para animar al consumo al cliente.

Hay dos cuestiones clave que deben ser señaladas cuando aplicamos el análisis asociativo a un conjunto de datos:

Descubrir patrones en grandes transacciones de conjuntos de datos puede ser costoso computacionalmente hablando.

Algunos de los patrones encontrados pueden ser falsos debido a que pueden haber ocurrido por casualidad.

A continuación veremos la terminología básica usada en el análisis asociativo.

Representación Binaria

Los problemas de análisis asociativo pueden ser representados utilizando atributos binarios. Un *item* se representará de forma binaria simplemente asignando un 1 si aparece en la transacción y un 0 en caso contrario. En la Tabla 2. Base de datos con atributos primarios se pueden observar los mismos datos que aparecen en la Tabla 1 pero representados mediante atributos binarios.

	PAN	LECHE	PAÑALES	CERVEZA	HUEVOS	COLA
1	1	1	0	0	0	0
2	1	0	1	1	1	1
3	0	1	1	1	0	1
4	1	1	1	1	0	0
5	1	1	1	0	0	1

Tabla 2. Base de datos con atributos primarios

Itemset y Frecuencia de ocurrencia

Sea I=i1, i2,…id un conjunto de ítems y T=t1,t2,…tn el conjunto de todas las transacciones almacenadas en una base de datos. Cada transacción t_i contiene un subconjunto de *ítems* de entre los posibles ítems definidos en I.

A toda colección de cero o más items se le llama *itemset*. Si un *itemset* contiene *k* ítems se le llama *k-itemset*. En el ejemplo de la cesta de la compra anterior la transacción {Pan, Leche, Cerveza, Huevos} es un ejemplo de un 4-itemset. Si tenemos un conjunto de datos con cero elementos también es válido y sería el *itemset* vacío.

El tamaño de una transacción lo definimos como el número de *items* presentes en la transacción. Una transacción t_j decimos que contiene un *itemset X* si *X* es un subconjunto de t_j,. Por ejemplo, la segunda transacción de la Tabla 2.2 contiene el *itemset* {Pan, Pañales} pero no contiene el *itemset* {Pan, Leche}.

Una propiedad que se asocia a un *itemset* es su frecuencia de ocurrencia que se define como el número de transacciones que contienen dicho *itemset*. La fórmula matemática de la frecuencia de ocurrencia está definida por la siguiente ecuación:

$$\sigma(X) = \left|\{t_i | X \subseteq t_i,\ t_i \in T\}\right|,$$

Para explicar mejor esta ecuación recurriremos como en ocasiones anteriores a la Tabla 2 en la que podemos ver que el *itemset* {Cerveza, Pañales, Leche} aparece en dos transacciones por lo que su frecuencia de ocurrencia es igual a 2.

Reglas de Asociación

Una regla de asociación es una expresión que tiene la forma $X \rightarrow Y$ donde X e Y son *itemsets* disjuntos.

Para medir la fuerza de una regla de asociación se utilizan los conceptos de *soporte* y *confianza***.** El *soporte* de una regla es la proporción de transacciones que contienen el *itemset* formado por los ítems que aparecen en el antecedente y consecuente de la regla. La *confianza* de una regla es la proporción de transacciones que conteniendo los ítems del antecedente de la regla también contienen los ítems del consecuente de la regla, es decir, la proporción de transacciones para las que se cumple la regla. Las definiciones formales que son las siguientes:

$$\text{Support, } s(X \longrightarrow Y) = \frac{\sigma(X \cup Y)}{N};$$
$$\text{Confidence, } c(X \longrightarrow Y) = \frac{\sigma(X \cup Y)}{\sigma(X)}.$$

Veamos un ejemplo de estos conceptos basado en los elementos de la cesta de la compra.

Consideremos la regla {Leche, Pañales} → {Cerveza}.Como sabemos que la frecuencia de ocurrencia de {Leche, Pañales, Cerveza} es 2 y el número total de transacciones es 5 el *soporte* de la regla es 2/5, es decir, el 40%. La *confianza* de la regla se obtiene dividendo la frecuencia de ocurrencia de {Leche, Pañales, Cerveza} entre la frecuencia de ocurrencia de {Leche, Pañales}. Como hay 3 transacciones que contienen leche y pañales la *confianza* para esta regla es 2/3. Es decir, la regla se cumple en el 66% de los casos.

El proceso de generación de reglas de asociación consiste en encontrar, para un conjunto de transacciones, todas las reglas que tengan un soporte y confianza superiores a unos valores mínimos a los que se denomina *minsup* y *minconf*, respectivamente. Este proceso puede descomponerse en 2 sub-tareas para facilitar su implementación.

1.-Generación de los itemsets frecuentes cuyo objetivo es descubrir todos los *itemsets* que satisfacen el umbral *minsup*. Estos *itemsets* descubiertos serán los denominados *itemsets* frecuentes.

2.-Generación de Reglas cuyo objetivo es extraer, a partir de los itemsets frecuentes encontrados en el paso previo, todas las reglas que tengan un grado de confianza alto.

En el siguiente apartado veremos la generación de *itemsets* frecuentes usando el algoritmo a priori [1].

2.2. Generación de itemsets frecuentes

Un conjunto de datos que contiene *k items* tiene la posibilidad de generar $2^k - 1$ itemsets frecuentes, excluyendo el conjunto nulo.

Como la *k* puede ser muy grande en muchas aplicaciones prácticas, el espacio de búsqueda de los *itemsets* es exponencialmente grande. Por ello, es necesario encontrar una manera de reducir el número de *itemsets* candidatos.

Una opción es usar el **algoritmo a priori**. Este algoritmo es una manera eficiente de eliminar algunos de los *itemsets* candidatos sin necesidad de contar sus valores de soporte**.**

El Principio A priori.
"Si un itemset es frecuente entonces todos sus subconjuntos deben ser también frecuentes" [1].

Para poder comprender mejor principio pondremos un ejemplo usando la Figura 1. Árbol de Transacción y subconjuntos. Supongamos que {c, d, e} es un *itemset* frecuente. Vemos claramente entonces que cualquier transacción que contenga {c, d, e} debe de contener también sus subconjuntos:{c, d},{c, e}, {d, e}, {c}, {d} y {e}.

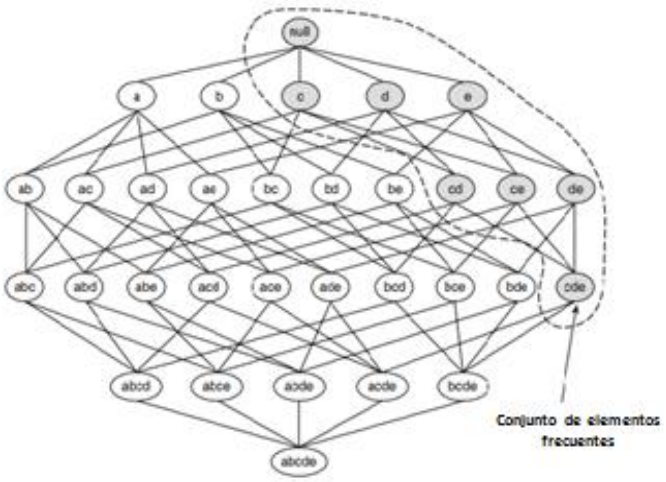


Figura 1. Árbol de Transacción y subconjuntos

Como contraposición si un *itemset* no es frecuente, tampoco lo son sus súper conjuntos. En la Figura 2. Árbol de poda de subconjuntos, si {a, b} no es frecuente, entonces todos sus súper conjuntos ({a, b, c}, {a, b, d},….) tampoco lo son.

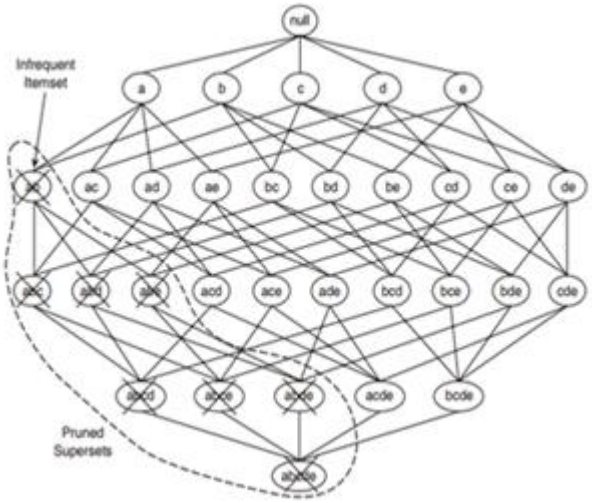


Figura 2. Árbol de poda de subconjuntos

Generación de *itemsets* frecuentes en el Algoritmo a priori

La Figura 3 muestra un ejemplo de la generación de *itemsets* frecuentes con el algoritmo a priori a partir de las transacciones mostradas en la Tabla 1. Base de datos con 5 transacciones.

Para comenzar con el ejemplo asumimos que tenemos un *soporte* umbral del 60%, que es equivalente a tener una frecuencia de ocurrencia mínima de 3.

Inicialmente, cada candidato es considerado como un 1-*itemset* candidato. Después de calcular los valores de soporte**,** los *itemsets* candidatos {Coca-Cola} y {Huevos} son descartados porque aparecen en menos de 3 transacciones que es frecuencia de ocurrencia mínima.

En la siguiente iteración generamos los 2-*itemsets* candidatos usando solo los 1-*itemsets* candidatos calculados anteriormente.

Como solo tenemos 4 1-*itemsets* candidatos el numero de 2-*itemsets* generados por el algoritmo será$\binom{4}{2}$n =6. Dos de esos 6 candidatos {Cerveza, Pan} y {Cerveza, leche} resultan ser poco frecuentes después de ver sus valores de frecuencia de ocurrencia.

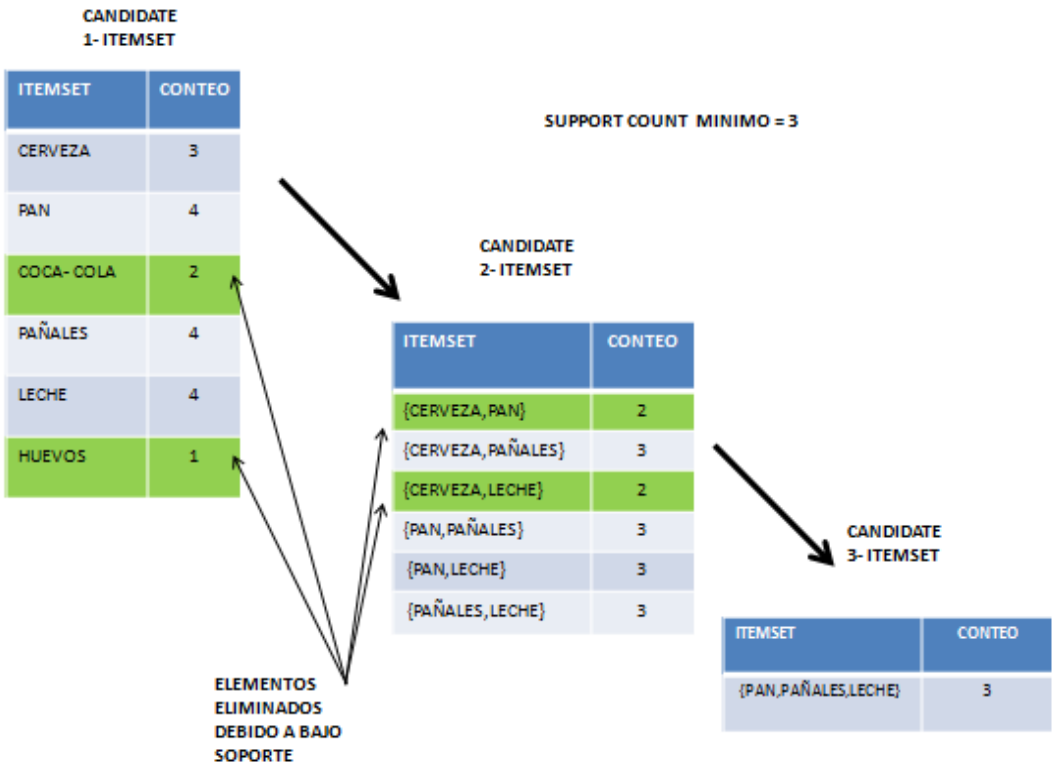


Figura 3. Ejemplo de generación de itemsets frecuentes

Los 4 candidatos restantes son frecuentes y serán los que usemos para generar los 3-*itemsets* candidatos.

Sin aplicar poda se pueden crear $\binom{6}{3}$=20 3-*itemsets* candidatos con los 6 items del ejemplo. Pero con el principio a priori solo tenemos que quedarnos con los 3-*itemsets* candidatos cuyos subconjuntos son frecuentes. El único candidato que cumple esa propiedad es {Pan, Pañales, Leche}.

Pseudocódigo para la generación de itemsets frecuentes

Para la generación de los *itemsets* frecuentes debemos de tener en cuenta una serie de pasos que describiremos a continuación. Los pasos a los que haremos referencia son las líneas del pseudocódigo que aparece en el Algoritmo 1.

- El algoritmo hace una pasada inicial sobre los datos para calcular la frecuencia de ocurrencia que tiene cada *ítem.* Cuando este paso termine, conoceremos el conjunto de los 1-*itemsets* frecuentes F_1 (paso 1 y 2).

- A continuación, el algoritmo genera iterativamente nuevos k-*itemsets* candidatos usando los *(k -1)-itemsets* frecuentes que hemos encontrado en la iteración previa (paso 5). La generación de candidatos se realiza con una función llamada a priori-gen, que describiremos en el apartado 2.3.

```
K=1
Fk= {i |i ∈ I ∧ σ({i}) ≥ N x minsup}  {Encuentra todos los i-itemsets}
repite.
   K = k+1.
   Ck = a priori-gen(Fk -1). {Genera los itemsets candidatos}
   para cada transaccion t ∈ T haz
     Ct = subconjunto(Ck, t) { identifica todos los candidatos que
pertenezcan a T}
      para cada itemset candidato c ∈ Ct haz
        σ(c) = σ(c) +1. {Incrementa el support count}
      termina para
    termina para
    Fk= {c |c ∈ Ck∧ σ({c}) ≥ N x minsup}    {Extrae los k-itemsets
frecuentes}
Hasta que Fk=0
Resultado= U Fk
```

Algoritmo 1. Generar K-Itemsets

- Para calcular la frecuencia de ocurrencia de los candidatos, el algoritmo necesita realizar una pasada adicional a todo el conjunto de datos (pasos 6-10). La función subconjunto se encarga de encontrar todos los *itemsets* candidatos en C_k que están contenidos en cada transacción t.

- Después de calcular sus frecuencias de ocurrencia, el algoritmo elimina todos los *itemsets* candidatos cuya frecuencia de ocurrencia es menor que el valor de *minsup* (paso 12).

- El algoritmo termina cuando no hay nuevos *itemsets* generados (paso 13) [2].

2.3 Generación y poda de candidatos

La función a-priori-gen de la que hemos hablado anteriormente genera *itemsets* candidatos realizando las siguientes dos operaciones:

- **Generación de Candidatos:** Esta operación genera nuevos k-*itemsets* candidatos basados en los (k -1) -*itemsets* frecuentes que hemos encontrado en la iteración previa.

- **Poda de Candidatos:** Esta operación elimina algunos de los k-*itemsets* candidatos usando la estrategia de poda basada en la frecuencia de ocurrencia

Existen varios métodos para generar candidatos. Nosotros vamos a utilizar el **método F_{k-1} x F_{k-1}** que vamos a explicar a continuación.

El método F_{k-1} x F_{k-1}

El procedimiento para generar candidatos en la función a priori-gen combina un par de *(k -1)-itemsets* frecuentes solo si sus primeros (k -2) *items* son iguales.

Supongamos que $A=a_1,a_2,\ldots\ldots a_{k-1}$ y que $B=b_1\ b_2,\ldots b_{k-1}$ son *un* par de (k-1)-*itemsets* frecuentes. A y B se combinarán si satisfacen las siguientes condiciones.

$$a_i = b_i \text{ (for } i = 1, 2, \ldots, k-2) \text{ and } a_{k-1} \neq b_{k-1}.$$

Como la definición formal puede resultar a veces confusa vamos a poner un ejemplo más práctico de cómo funciona este método usando la Figura 4.

En la Figura 4. Procedimiento de generación de candidatos los *itemsets* frecuentes {Pan, Pañales} y {Pan, Leche} se combinan para formar un candidato 3-*itemset* {Pan, Pañales, Leche}.

El algoritmo no puede combinar {Cerveza, Pañales} con {Pañales, Leche} porque el primer *item* de cada uno de los *itemsets* es diferente.

De hecho si {Cerveza, Pañales, Leche} fuese un candidato viable, habría sido generado con la combinación de {Cerveza, Pañales} y {Cerveza, Leche}. Con este ejemplo lo que queremos ilustrar es que el procedimiento de generación de candidatos es el más completo, y señalar también las ventajas de usar el ordenamiento lexicográfico para prevenir los candidatos duplicados.

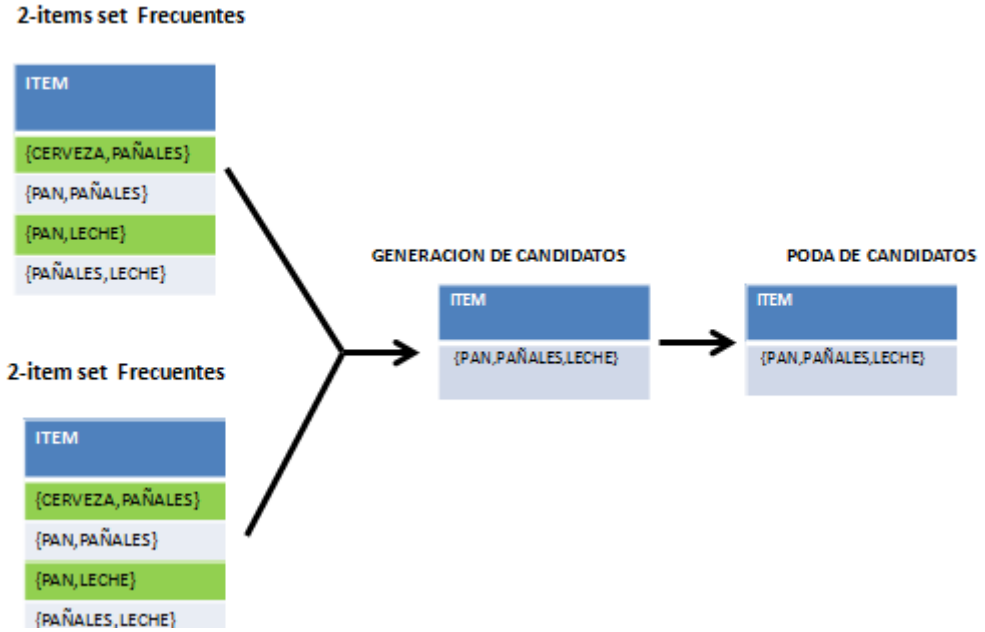


Figura 4. Procedimiento de generación de candidatos

2.4 Cálculo de las frecuencias de ocurrencia

El cálculo de frecuencias de ocurrencia es el proceso usado para determinar el número de apariciones de un *itemset* que ha sobrevivido a la poda en la función a priori-gen.

Un ejemplo que nos ayudará a comprender lo explicado anteriormente es el siguiente. Vamos a considerar que tenemos una base de datos que contiene las siguientes transacciones {1,2,3,4} , {1,2,3,4,5} , {2,3,4} , {2,3,5} , {1,2,4} , {1,3,4} , {2,3,4,5} , {1,3,4,5} , {3,4,5} , {1,2,3,5} donde cada número corresponde a un *item* de la transacción. Primero debemos de calcular la frecuencia de ocurrencia de cada item por separado obteniendo los resultados que aparecen en la Figura 5. Frecuencias de ocurrencia[3].

ELEMENTO	APOYO
1	6
2	7
3	9
4	8
5	6

Figura 5. Frecuencias de ocurrencia

Si definimos una frecuencia mínima de 4, todos los elementos son frecuentes y no podamos ninguno. En el siguiente paso seleccionamos sólo los (*2-itemsets*) que son frecuentes, es decir, aquellos cuya frecuencia de ocurrencia es mayor o igual a 4. En la Figura 6. Frecuencias de ocurrencia de los 2-itemsets, eliminaríamos el itemset {1,5}.

2-itemsets	FRECUENCIA DE OCURRENCIA
{1,2}	4
{1,3}	5
{1,4}	5
{1,5}	3
{2,3}	6
{2,4}	5
{2,5}	4
{3,4}	7
{3,5}	6
{4,5}	4

Figura 6. Frecuencias de ocurrencia de los 2-itemsets

Por último se genera la lista de los 3- *itemsets* frecuentes. El resultado se muestra en la Figura 7. Lista de los 3-triples items frecuentes .

ELEMENTO	APOYO
1	6
2	7
3	9
4	8
5	6

Figura 7. Lista de los 3-triples items frecuentes

El algoritmo terminará aquí porque el par {2, 3, 4, 5} que generaríamos en el siguiente paso no tendría la frecuencia de ocurrencia mínima.

Ahora vamos a aplicar el mismo algoritmo al mismo conjunto de datos pero cambiando el valor de minsup a 5. Obtenemos los siguientes resultados:

Paso 1:

ELEMENTO	APOYO
1	6
2	7
3	9
4	8
5	6

Figura 8. Lista de los 3-triples items frecuentes

Paso 2:

2 -itemsets	FRECUENCIA DE OCURRENCIA
{1,2}	4
{1,3}	5
{1,4}	5
{1,5}	3
{2,3}	6
{2,4}	5
{2,5}	4
{3,4}	7
{3,5}	6
{4,5}	4

Figura 9. Lista de todos los 2 itemset frecuentes

El algoritmo termina aquí porque ninguno de los 3-*items* generados en el paso 3 tiene el *minsup* deseado.

2.5 Cálculo de las frecuencias de ocurrencia mediante hash trees

Aparte de la forma anteriormente explicada, hay otra forma más eficiente de realizar el *cálculo de las frecuencias de ocurrencia*. Podemos usar un HashTree, estos árboles consisten un nodo padre del cual cuelgan nodos hijos que a su vez pueden convertirse en nodos padres con sus consiguientes nodos hijos.

Usando un árbol Hash podríamos guardar los *itemsets* candidatos en diferentes hojas de el árbol hash por lo cual las comparaciones se hacen por hojas con la consecuente ganancia de tiempo.

Mediante el algoritmo que los *itemsets* contenidos en cada transacción se introducen en las hojas que les correspondan de la tabla hash. De esta manera en vez de comparar cada *itemset* de la transacción con cada *itemset* candidato, se compara con los *itemsets* candidatos que pertenecen a la misma hoja como mostramos en la Figura 10. Contador de apoyo de elementos empleando estructuras hash.

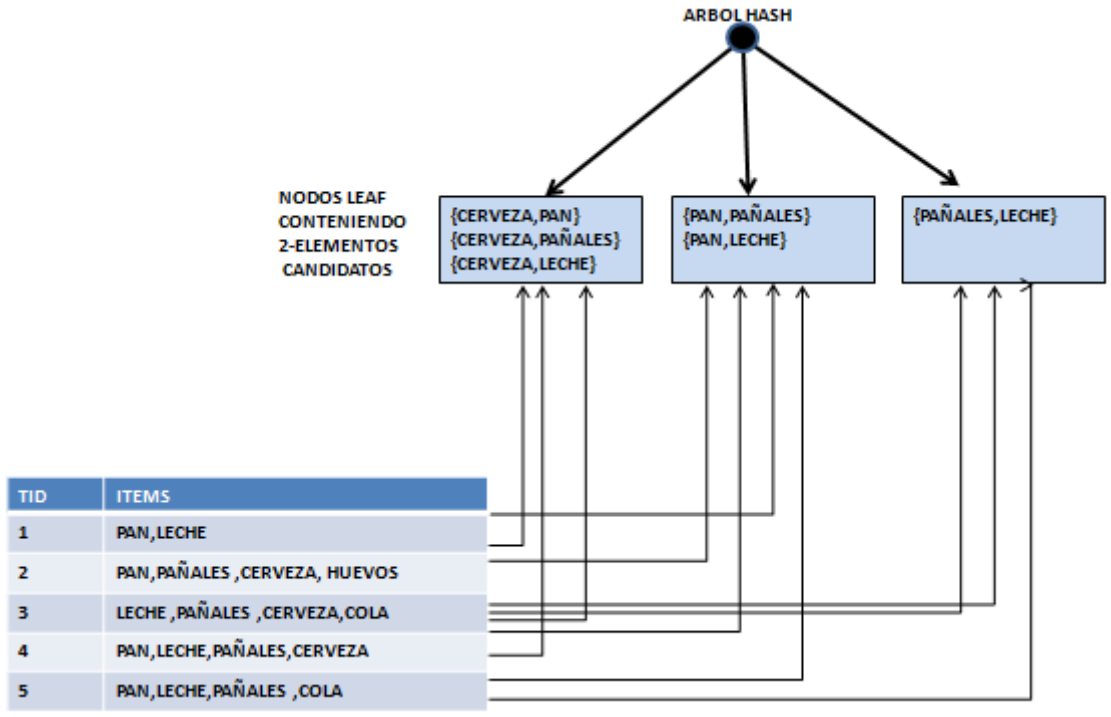


Figura 10. Contador de apoyo de elementos empleando estructuras hash

Como la explicación teórica puede no ser lo suficientemente clara, a continuación pondremos una explicación más práctica para ver cómo se comporta este algoritmo.

Supongamos que queremos calcular las frecuencias de ocurrencia para unos Ck itemsets.

Con la nueva estructura de hash tree lo primero que debemos hacer es crear el hash tree e introducir los k itemsets candidatos en los nodos de las hojas del árbol.

Para cada transacción generamos todos los subconjuntos de k-itemsets de la transacción. Por ejemplo para la transacción {1, 2 , 3, 4} los conjuntos de items son {1 , 2 , 3} , {1 , 2 , 4} , {1 , 3 , 4} y {2 , 3 , 4}.

Introducimos cada subconjunto de k-item en un nodo hoja del hash tree, y comprobamos con todos los k-itemsets candidatos que habíamos metido en el mismo nodo hoja. Si el subconjunto de k-item encuentra un k-itemset candidato incrementa la frecuencia de ocurrencia de ese k-itemset candidato.

2.6 Generación de reglas

En este apartado vamos a describir como extraer las reglas asociativas de una forma eficiente para un *itemset* frecuente dado. Cada *itemset frecuente* Y puede generar $2^k - 2$ reglas de asociación, ignorando las reglas que tienen sus antecedentes o consecuentes vacíos ($0 \rightarrow Y$ o $Y \rightarrow 0$).

Una regla de asociación puede ser extraída particionando el *itemset* Y en dos subconjuntos no vacíos, X y Y – X, de tal manera que $X \rightarrow Y - X$ satisface la confianza umbral. Hay que tener en cuenta que todas estas reglas satisfacen el *soporte* umbral puesto que han sido generadas a partir de un *itemset frecuente.*

GENERACION DE REGLAS PARA EL ALGORITMO A PRIORI

El algoritmo a priori usa un nivel de enfoque prudente para generar las reglas de asociación, donde cada nivel corresponde al número de *items* que pertenecen a la regla consecuente.

Inicialmente, todas las reglas con alta confianza que solo tienen un *item* en el consecuente son extraídas. Estas reglas son usadas para generar nuevas reglas candidatas.

Por ejemplo, si {acd} → {b} y {abd} → {c} son reglas con una alta confianza, entonces la regla candidata {ad} → {bc} se genera con la combinación de los consecuentes de las dos reglas anteriores.

La Figura 11. **Poda de Reglas** muestra una estructura de grafo para las reglas de asociación generadas por el *itemset frecuente* {a, b, c, d}. Si algún nodo de la celosía tiene menos confianza, todo el sub-grafo abarcado por el nodo puede ser podado inmediatamente. Supongamos que la confianza para {bcd} → {a} es baja. Todas las reglas que contengan el *item* a en sus consecuentes, incluyendo {cd} → {ab}, {bd} → {ad} y {d} → {abc} pueden ser descartadas [4].

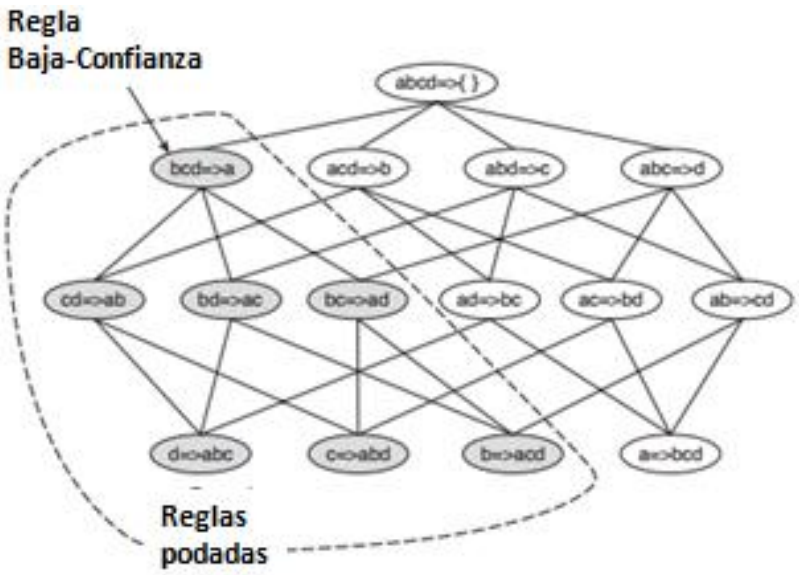


Figura 11. Poda de Reglas

Los Algoritmo 2. Algoritmo para la generación de reglas de asociación y Algoritmo 3. Algoritmo para la función ap-generareglas muestran las funciones para la generación de reglas.

para cada k-itemst frecuente tal que $f_k, k \geq 2$ **haz**
  $H_1 = \{ i \mid i \in f_k \}$ {1-item consecuencia de la regla}
  llama ap-generareglas(f_k, H_1)
termina para.

Algoritmo 2. Algoritmo para la generación de reglas de asociación

k= $|f_k|$ {tamaño del itemset frecuente}
m=$|H_m|$ {tamaño de la regla consecuente}
si k > m+1 **entonces**
  H_{m+1} = a priori-gen (H_m)
  para cada $h_{m+1} \in H_{m+1}$ **haz**
    con f =$\sigma(f_k) / \sigma(f_k - h_{m+1})$
    si con f $\geq$ minconf **entonces**
     devuelve la regla $(f_k - h_{m+1}) \rightarrow h_{m+1}$
    sino
     borra h_{m+1} de H_{m+1}
    termina si
  termina para
llama ap-generareglas(f_k, H_{m+1})
termina if

Algoritmo 3. Algoritmo para la función ap-generareglas

CAPITULO 3
MAP REDUCE

3.1 ¿Que es MapReduce?

MapReduce es un *framework* diseñado por Google en 2004 para procesar grandes conjuntos de datos de forma paralela en varias máquinas.

El procesamiento con MapReduce consta de dos funciones. La primera función llamada Map toma un conjunto de datos y los transforma en un par <clave, valor>. A continuación, la función Reduce toma ese par <clave, valor> y agrupa todos los pares que encuentre con la misma clave. Esa agrupación sería el resultado de la función MapReduce.

En general el concepto es simple, pero puede adquirir mayor complejidad si consideramos que:

- Al menos todos los datos deben ser traducidos a pares <clave, valor>.
- Las claves y valores podrían ser de cualquier tipo: cadenas, enteros… y, por supuesto pares <clave, valor> en sí mismos.

Un ejemplo del flujo de trabajo de MapReduce lo podemos encontrar en la Figura 12. Flujo de trabajo de MapReduce

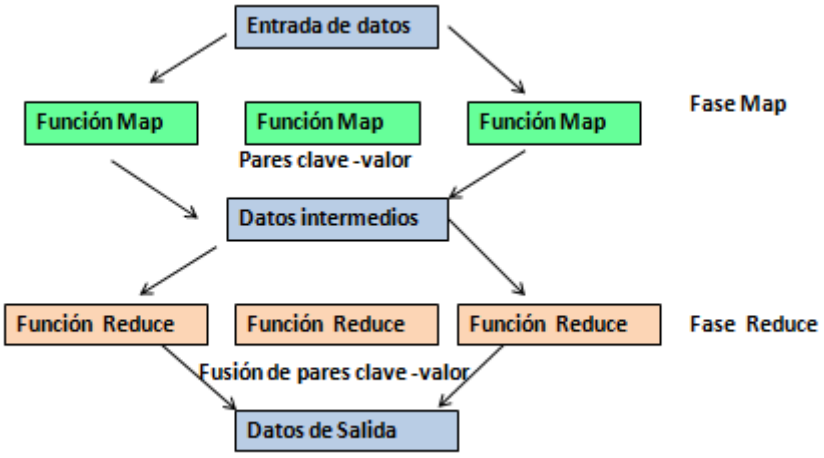


Figura 12. Flujo de trabajo de MapReduce

3.2 Proceso de MapReduce

Para aplicar MapReduce a un conjunto de datos es necesario crear las dos funciones anteriormente indicadas: la *función* Map y la *función* Reduce. El resto será manejado por el *framework* Amazon Elastic MapReduce (EMR). Amazon EMR es un servicio web que facilita el procesamiento rápido y rentable de grandes cantidades de datos. Amazon EMR simplifica el procesamiento de Big Data, al proporcionar un marco de trabajo Hadoop gestionado que facilita la distribución y el procesamiento de grandes cantidades de datos entre instancias de Amazon EC2 dinámicamente escalables de manera sencilla, rápida y rentable [5].

Cuando lancemos el proceso MapReduce el conjunto de datos de entrada se dividirá en segmentos, pasando cada segmento a una máquina diferente para su análisis. Cada máquina ejecutará la función Map sobre la parte del conjunto de datos que le haya sido asignada.

La función Map (que hayamos creado) tomará el conjunto de los datos de entrada y los transformará en un par de <clave, valor> según las especificaciones que deseemos.

Por ejemplo, si lo que buscamos es contar frecuencias de palabras en un texto, tomaríamos el par <palabra, contador> como nuestro par de <clave, valor>, y la función Map se encargaría de crear un par < palabra, 1 > para cada palabra que encontrase en el fichero de entrada.

Hay que tener en cuenta que la función Map no cuenta la frecuencia con la que aparecen las palabras en el texto de entrada. Para lograr esto usaremos la función Reduce.

Los pares <clave, valor> que tengan la misma clave se agrupan y se pasan a una sola máquina que ejecutará sobre ellos la función Reduce.

La función Reduce obtiene una colección de pares <clave, valor> y los "reduce" usando para ello el algoritmo específico que hayamos creado.

En nuestro ejemplo, queremos contar las veces que aparece una palabra en el texto para poder obtener la frecuencia. Para ello nuestra función Reduce simplemente sumará los valores de los pares <clave, valor> que tengan la misma clave.

La Figura 13. Ejemplo de wordcount describe el escenario que hemos definido.

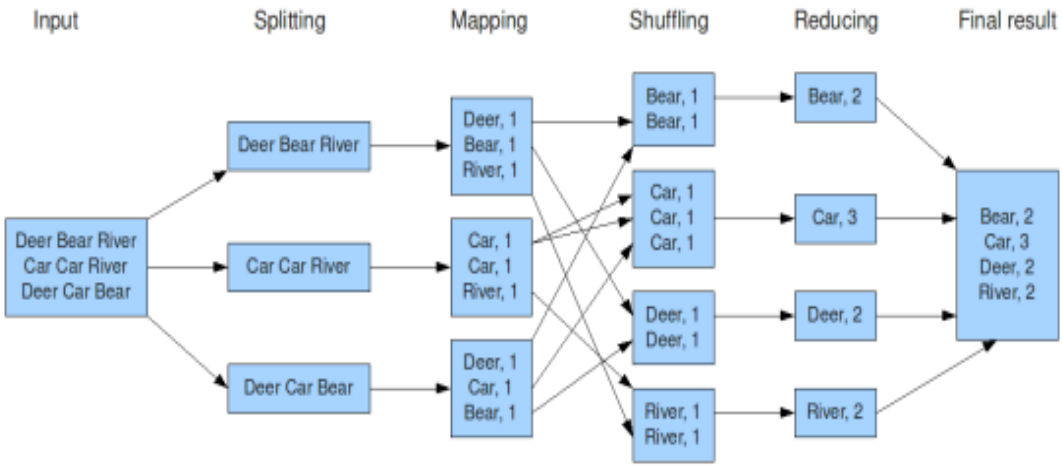


Figura 13. Ejemplo de wordcount

El ejemplo del recuento de palabras que hemos usado, es el más sencillo que existe de MapReduce, pero MapReduce es mucho más eficiente que eso, a continuación vemos unos ejemplos de lo que se puede obtener usando MapReduce.

- Ordenación distribuida
- Búsqueda distribuida
- Grafos de enlaces web distribuida
- Aprendizaje automático

Para que la idea de MapReduce quede más completa mostraremos a continuación un ejemplo con imágenes del flujo de datos que seguiría MapReduce para crear el algoritmo de wordcount explicado anteriormente.

Lo primero que hace MapReduce es dividir el fichero del wordcount en bloques como podemos ver en la Figura 14. División en bloquesCada bloque se pasa a una tarea Map (Map Task).

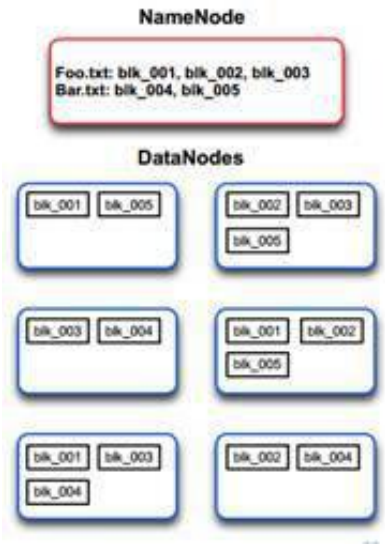


Figura 14. División en bloques

A continuación mostraremos de forma ilustrada la ejecución del algoritmo wordcount con MapReduce. Usaremos de entrada un fichero cuyas líneas tienen como clave el número de línea y como valor la cadena de caracteres. Podemos observar un ejemplo en la Figura 15. Entrada a la función Map

```
(3414, 'the cat sat on the mat')
(3437, 'the aardvark sat on the sofa')
```

Figura 15. Entrada a la función Map

La función Map implementaría el algoritmo mostrado en la Figura 16. Función Map que convertiría la entrada en pares de <palabra, contador>.

```
map(String input_key, String input_value)
    foreach word w in input_value:
        emit(w, 1)
```

Figura 16. Función Map

Este algoritmo generaría una salida de la forma en que se ve en la Figura 17. Salida de la función Map.

```
('the', 1), ('cat', 1), ('sat', 1), ('on', 1),
('the', 1), ('mat', 1), ('the', 1), ('aardvark', 1),
('sat', 1), ('on', 1), ('the', 1), ('sofa', 1)
```

Figura 17. Salida de la función Map

Una vez obtenida la salida de la función Map el framework agrupa los datos con la misma clave, de tal forma que se obtienen tantos pares (clave, valor) como tareas Reduce se ejecutarán.

En nuestro caso cada tarea Reduce sumará los valores de entrada y generará una única salida con la palabra y su frecuencia de aparición en el texto.

A cada tarea Reduce le llegaría una entrada como la que vemos en la, Figura 18. Entrada a la función Reduce

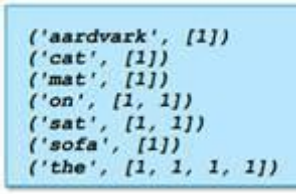


Figura 18. Entrada a la función Reduce

A esta entrada le será aplicado el algoritmo de suma que habremos creado en nuestra función Reduce. Un pseudocódigo de este algoritmo lo encontramos en la Figura 19. Función Reduce

```
reduce(String output_key,
                    Iterator<int> intermediate_vals)
    set count = 0
    foreach v in intermediate_vals:
        count += v
    emit(output_key, count)
```

Figura 19. Función Reduce

Una vez aplicado el algoritmo a los pares de entrada, la función Reduce nos generará una salida como la de la Figura 20. Salida de la función Reduce [6].

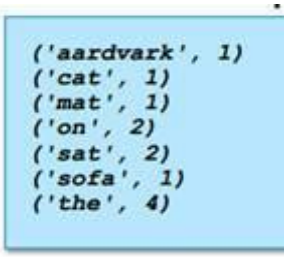


Figura 20. Salida de la función Reduce

3.3 Ejecución Hadoop

Hadoop es la implementación gratuita y de código abierto de Apache del algoritmo MapReduce (a diferencia, por ejemplo, de la aplicación patentada de Google). El flujo de trabajo EMR es el que usa Hadoop para el framework MapReduce.

Una vez creado el MapReduce se deberá proceder a la creación de cuatro elementos para poder ejecutar el programa. Estos elementos son el NameNode, DataNode, el JobTracker y el TaskTracker.

La Figura 21. Ejecución Hadoop muestra el comportamiento de la ejecución de un programa en Hadoop.

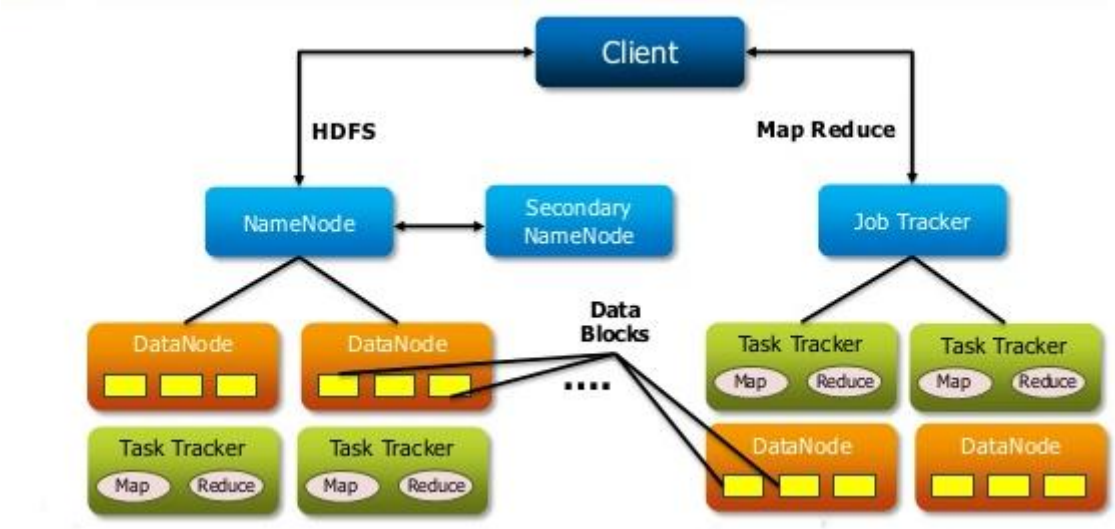


Figura 21. Ejecución Hadoop

NAMENODE Y DATANODE

El NameNode es el maestro de todos los DataNode. Es el responsable de manejar el namespace del sistema de ficheros y controlar los accesos de los clientes externos. Esta información la almacena en un fichero llamado FsImage junto a otro fichero llamado EditLog, que almacena las transacciones de HDFS en el sistema de ficheros local del nodo. Además esta información se replica para protegerlo contra posibles pérdidas o corrupciones del sistema de ficheros. Los metadatos contenidos en el NameNode indican en cual o cuales DataNodes está almacenado cada objeto y procesa las operaciones sobre el sistema de ficheros determinando que DataNode es el responsable de ejecutarla. Cada bloque de un objeto se guarda en diferentes DataNodes para proporcionar redundancia.

Normalmente se ejecuta en una máquina separada del resto de nodos, pero no tiene porqué ser obligatoriamente así.

Haciendo una abstracción, podemos decir que asume el rol de una tabla de ficheros en un sistema de ficheros tradicional (como FAT, MFT o la tabla de inodos en FAT32, NTFS y ext3/ext4 respectivamente) o de un DNS en Internet. Almacena la información relativa al lugar donde se encuentran los objetos en HDFS.

Este tipo de nodos actúa como un almacenamiento de bloques para HDFS. Un clúster de Hadoop puede contener cientos o miles de DataNodes. Estos DataNodes responden a las peticiones de lectura y escritura de los clientes y hacen réplicas de los bloques que les envía el NameNode.

Cada DataNode envía periódicamente paquetes al NameNode con información de sus bloques para que éste pueda validar la consistencia de la información con otros DataNodes.

Normalmente todos los DataNodes están en un rack conectados a un switch común para aumentar el ancho de banda y el rendimiento del clúster.

JOBTRACKER

El JobTracker es un servicio dentro de Hadoop que lleva las tareas MapReduce a nodos específicos del clúster, en el mejor de los casos, a nodos que tienen los mismos datos, o que por lo menos están en el mismo rack. Su funcionamiento es el siguiente:

1º.- Las aplicaciones clientes envían trabajos al JobTracker.

2º.- El JobTracker se comunica con el nodo para ver la localización de los datos.

3º.- El JobTracker localiza nodos TaskTracker con espacio disponible (llamado ranuras) disponibles en o cerca de los datos.

4º - El JobTracker envía el trabajo a los nodos TaskTracker elegidos.

5º.- Los nodos TaskTracker son monitorizados. Si no envían señales de latido con la suficiente frecuencia, se considera que han fracasado y el trabajo se reenvía a un TaskTracker diferente.

6º.- En el caso de que una tarea falle, el TaskTracker se lo notificará al JobTracker. El JobTracker decidirá qué hacer a continuación. Puede volver a presentar el trabajo en otros lugares, puede marcar ese registro específico como algo a evitar, o puede poner en la lista de poco fiables al TaskTracker.

7º.- Cuando el trabajo ha sido completado, el JobTracker actualiza el estado del mismo.

8º.- Las aplicaciones clientes pueden sondear el JobTracker en busca de información.

TASKTRACKER

Un TaskTracker es un nodo del clúster que acepta tareas MapReduce y operaciones aleatorias desde un JobTracker.

Cada TaskTracker está configurado con una serie de ranuras que indican el número de tareas que puede aceptar. Cuando el JobTracker trata de encontrar un lugar para programar una tarea dentro de las operaciones de MapReduce, primero busca un espacio vacío en el mismo servidor que aloja el DataNode que contiene los datos, y, si no lo encuentra busca otro en una máquina en el mismo rack.

El TaskTracker genera unos procesos JVM separados para hacer el trabajo real. Esto es para asegurase de que un fallo en el proceso no acaba con el TaskTracker. El TaskTracker supervisa estos procesos generados, capturando los códigos de salida. Cuando el proceso termina el TaskTracker se lo notifica al JobTracker. Los TaskTrackers también envían mensajes de latido al JobTracker, generalmente cada pocos minutos, informando al JobTracker de que todavía está activo. Estos mensajes también informan al JobTracker del número de plazas disponibles, por lo que el JobTracker puede mantenerse al día con los lugares en los que el clúster puede delegar información [7].

CAPITULO 4
IMPLEMENTACIÓN DE LOS ALGORITMOS

Existen dos formas de implementar el algoritmo "A Priori". Podemos realizarlo tanto usando el modelo de datos HashTree, como usando el modelo de datos HashSet. Con la finalidad de seleccionar uno de los dos, primero se comparan los tiempos de procesamiento, eligiéndose para continuar con el resto del proyecto el que tenga un tiempo más bajo.

Para entender qué es un HashTree o un HashSet antes deberemos explicar dos términos que son imprescindibles.

El primero de los términos es el árbol de decisión. Los árboles de decisión se utilizan para asignar instancias a una clase concreta. En un árbol de decisión cada nodo es un atributo y cada rama representa un valor posible de ese atributo. Los nodos hoja representan las clases.

Un ejemplo del funcionamiento de un árbol de decisión lo podemos ver en la Figura 22. Árbol de Decisión. Como se observa, en el nodo padre se realiza la pregunta de si es o no de sexo masculino. En caso de que la respuesta sea no vamos a la rama final y el resultado es que sobrevive. Si la respuesta es sí, en el siguiente nodo se realiza una pregunta de si su edad es mayor de 9.5 años o no. Si la respuesta es afirmativa se va a la rama final con el resultado de que muere. Si por el contrario la respuesta es negativa, se realiza en el siguiente nodo la pregunta de si tiene más de 2.5 hermanos. Si la respuesta es afirmativa el resultado de la rama final es que muere, en caso contrario es que sobrevive [8].

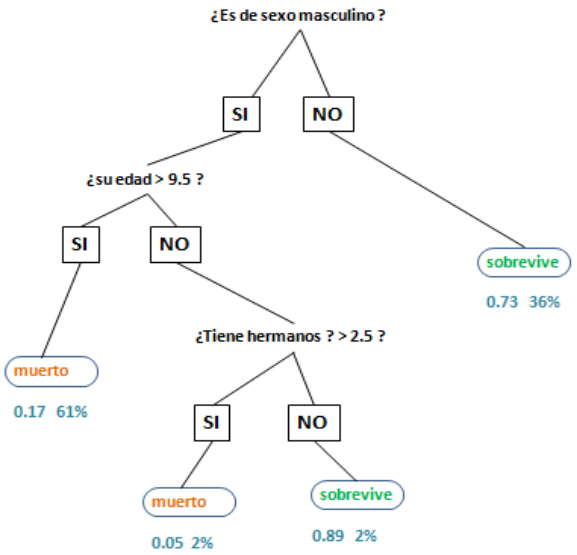


Figura 22. Árbol de Decisión

Una tabla hash, es una estructura de datos que asocia *claves* con *valores*. La operación principal que soporta de manera eficiente es la *búsqueda*. Esta tabla permite el acceso a los elementos almacenados a partir de una clave generada. Funciona transformando la clave con una función hash en un *hash* (número que identifica la posición de la tabla hash localiza el valor deseado).

En la Figura 23. Tabla Hash podemos observar el funcionamiento de una tabla hash. A partir de la clave de la izquierda obtenemos un valor *hash* que será su posición en la tabla. Esta posición será la que usemos para la búsqueda [9].

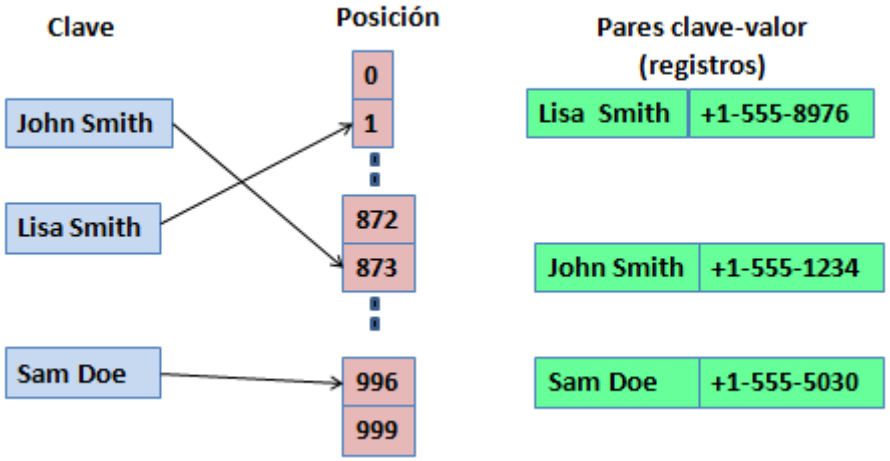


Figura 23. Tabla Hash

El primer algoritmo que ha sido implementado lo ha sido con la estructura de datos HashTree. Un hash tree es un árbol en el que cada nodo interno está etiquetado con el valor hash de las etiquetas de sus nodos hijo.

La Figura 24. HashTree donde puede apreciarse que el nodo hash 0 que cuelga del nodo raíz es el resultado de aplicar la función hash a la concatenación de hash 0-0 y hash 0-1.

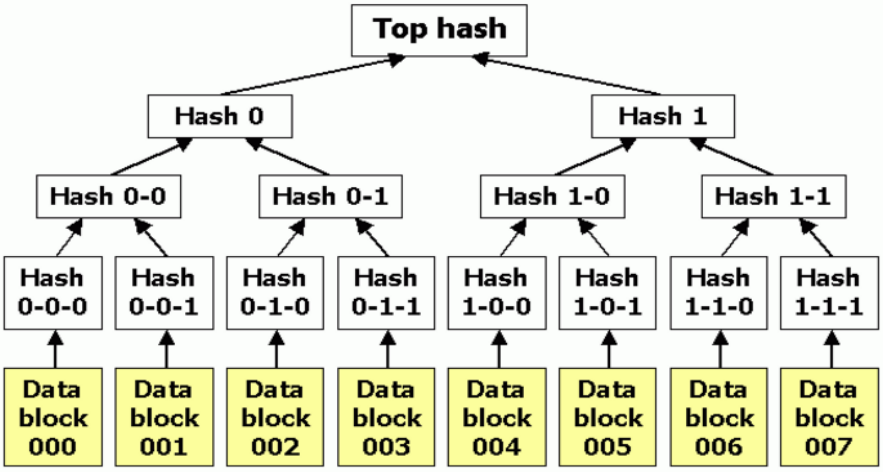


Figura 24. HashTree

En nuestro caso, al insertar los *itemsets* por transacciones, el propio árbol de la estructura de datos HashTree, los agrupa en función los datos de las mismas transacciones. Esto hace mucho más eficiente la búsqueda de dichos *itemsets*, para la generación de candidatos.

Para este algoritmo, creado con la estructura de datos HashTree, hemos usado como referencia un código de la Universidad de Regina **[10]**.

El segundo tipo de algoritmo hace uso de las estructuras de datos HashSet. Una estructura de datos HashSet consiste en una tabla Hash como las explicadas con anterioridad, sobre la que no se permite insertar elementos duplicados en ningún caso. Esta estructura de datos es más eficiente que la anterior, y por ello será la escogida, puesto que al no permitir *itemsets* duplicados, la generación de candidatos será más eficiente.

Para este algoritmo hemos usado como referencia un algoritmo de Google Docs **[11]**.

4.1 Implementación Hashtree

Como podemos ver en el algoritmo de referencia, en este se realizaba la creación de candidatos e *itemsets* frecuentes obteniendo un vector de candidatos sobre el que después se generarían los *itemsets* frecuentes usando las fórmulas matemáticas y los algoritmos que hemos estudiado en el capítulo 2.

Esta solución es bastante ineficiente puesto que se han de recorrer los vectores enteros para comparar *ítems* y seleccionar los candidatos, y una vez obtenidos estos se tiene que hacer lo mismo para conseguir los *itemsets* frecuentes. Por ello, y usando lo explicado en el capítulo 2 sobre el uso de HashTree en el algoritmo "A Priori", decidimos cambiar ese sistema de vectores por uno de HashTree. El cambio es sencillo: creamos un árbol Hash que será el padre. Este árbol Hash tiene un identificador para saber el nodo en el que se encuentra y está compuesto de árboles Hash que representarían sus hojas.

En cada transacción iremos sacando los candidatos, y agrupándolos en el mismo nodo hoja para que cuando queramos obtener los *itemsets* frecuentes solo tengamos que acudir al nodo apropiado.

4.2 Implementación Hashset

Con este algoritmo se generarán los candidatos, y los *itemsets* frecuentes, siguiendo las fórmulas matemáticas y los algoritmos del capítulo 2. Existe sin embargo una diferencia- En este caso se utilizará un HashSet como los explicados anteriormente, y puesto que este no permiten duplicados tendremos una lista de candidatos únicos que nos generará los *itemsets* frecuentes de forma mucho más sencilla. Esto se debe a que en las distintas iteraciones del algoritmo no se deberá comprobar que pueda haber *itemsets* repetidos, por lo que no deberemos recorrer de forma iterativa ningún tipo de datos, o hacer comparaciones, con la pérdida de tiempo de procesamiento que esto conlleva.

Para hacer este algoritmo más eficiente hemos realizado unos cambios con respecto a la referencia utilizada. Creamos una función que nos daba la frecuencia de aparición de los candidatos, así que cuando los íbamos generando, si el candidato ya aparecía, se incrementaba la variable de frecuencia, con lo cual la generación de *itemsets* frecuentes se convertía en algo mucho más trivial dado que anteriormente había que buscar en las transacciones los candidatos para ver su número de apariciones.

Por último convertimos en variables globales tanto los HashSet que contenían los candidatos como los *itemsets* frecuentes. Esto se hizo así para no tener que referenciarlos en las funciones y de esta forma, agilizar el uso de estos conjuntos de datos.

4.3 MapReduce

Una vez explicado el funcionamiento de los algoritmos que generan las reglas de asociación, deberemos crear los algoritmos MapReduce para poder aplicar el procesamiento en paralelo, esto es la ejecución del algoritmo en pequeños trozos de forma simultánea que hace que el tiempo de procesamiento sea menor.

Como hemos visto en el capítulo 3, la clase Map es la primera que ha de generarse, ya que es la que recibe el fichero de entrada, y aplica un algoritmo que convierte los datos en la forma <clave valor> para pasarlos a función Reduce. La función Map en nuestro caso se encargará de generar los candidatos, y enviarlos a la función Reduce de la forma explicada con anterioridad.

Lo primero que hace, es tratar el fichero para obtener las transacciones. A continuación se le aplica la función diseñada para obtener los candidatos, y esta lista de candidatos generados para las transacciones encontradas en el fichero se itera para ir enviando los resultados a la función Reduce.

Debido a que la ejecución es en paralelo, esta función se ejecutará múltiples veces dependiendo de los nodos que tengamos.

La Figura 25. Implementación de la clase Map muestra nuestra implementación de la clase Map.

```
public class MapFunction extends MapReduceBase  {
    Set<Set<String>> largeItemSet=new HashSet<Set<String>>();
    Set<Set<String>> currentItemSet=new HashSet<Set<String>>();

    /*Maps the file into memory */
    Map<Integer, Set<String>> fileMap=new HashMap<Integer, Set<String>>();
    Apriori a=new Apriori();
    public void map(LongWritable key, Text values,
            MapWritable context) throws IOException {

            a.filemap(values.toString());
            currentItemSet=a.currentItemSet();

             Iterator<Set<String>> litr =  currentItemSet.iterator();
             while(litr.hasNext()){
                context.keySet().add((Writable) litr);
             }
    }

}
```

Figura 25. Implementación de la clase Map

Una vez finalizada la ejecución de todos los Maps en paralelo, la clase Reduce comenzará a ejecutarse. Ésta recibe los candidatos obtenidos por la función Map y los introduce en una variable del tipo HashSet. Una vez obtenidos los datos se generan los *itemsets frecuentes* en otro HashSet. Cuando la función *Reduce* ha terminado de generar los *itemsets frecuentes* éstos se envían a la salida usando para ello un bucle iterativo y la función write.

La Figura 26. Implementación de la clase Reduce nuestra implementación de la clase Reduce.

```
public  class ReduceFunction extends MapReduceBase  {
        Set<Set<String>> currentItemSet=new HashSet<Set<String>>();
        Set<String> valores= new HashSet<String>();
        /*Maps the file into memory */
        Map<Integer, Set<String>> fileMap=new HashMap<Integer, Set<String>>();
        Apriori a;
        private DataOutput retorno;
    public void reduce(MapWritable value,Writable salida) throws IOException {
            valores.add(value.keySet().toString());
            currentItemSet.add(valores);
            currentItemSet = a.getNextLevelItemSet(currentItemSet);
            ArrayList<LargeItemSetVO> set = new ArrayList<LargeItemSetVO>(DataHelper.getLargeItemSetWithSupport());
            Collections.sort(set);
            for(LargeItemSetVO vo: set) {
                String cadena=vo.getItems().toString() + ", " + vo.getSupport()*100+"%" ;
                retorno.writeChars(cadena);
                 salida.write(retorno);
            }

    }
}
```

Figura 26. Implementación de la clase Reduce

Cuando se han terminado de ejecutar todas las funciones *Reduce* se mostrarán en un fichero todos los *itemsets frecuentes* generados, y el tiempo que han tardado en ejecutarse las funciones. También se incluirá el log de la aplicación para ver cómo se ha ido generando.

A continuación, en la Figura 27. Salida MapReduce mostramos un ejemplo de la salida de la aplicación. Esta imagen muestra el resultado de la ejecución del algoritmo MapReduce. Como podemos observar cada línea es una transacción con la regla generada y la confianza y el soporte mínimo de la misma. La última línea es el tiempo total de ejecución.

```
[1, 0, P] => [] (conf:100.0% supp:11.445% )
[6] => [] (conf:93.68183527641969% supp:12.455% )
[1, 1B] => [] (conf:100.0% supp:10.22% )
[2, 0, NL] => [] (conf:100.0% supp:12.205% )
[1, P] => [] (conf:100.0% supp:18.935% )
[3, 0] => [] (conf:100.0% supp:14.49% )
[5] => [] (conf:93.17768344451184% supp:15.365% )
[0, P] => [] (conf:100.0% supp:12.02% )
Execution time is: 5.21 seconds.
```

Figura 27. Salida MapReduce

CAPITULO 5
PRUEBAS

Una vez que hemos desarrollado tanto el algoritmo "A Priori" como los scripts de las funciones map y del reduce, procederemos a diseñar una batería de pruebas para ver su correcto funcionamiento.

Para comenzar la batería de pruebas se crean dos carpetas, una para almacenar los ficheros de entrada (in) y otra para almacenar el fichero de salida del algoritmo (out).

La carpeta de entrada contendrá archivos "csv" con las transacciones que queremos analizar. Para poder comprobar la funcionalidad de nuestros algoritmos con respecto al tamaño de los datos de entrada se han creado varios ficheros "csv" con un número creciente de transacciones. En la carpeta de salida se guardarán los datos una vez acabado el procesamiento. Estos datos comprenden básicamente dos archivos, uno con el log de proceso y otro con el resultado final en formato de texto.

5.1 Objetivos de las Pruebas

Los objetivos de este test de pruebas son los siguientes:

- Dimensionar el rendimiento y la escalabilidad del HDFS de hadoop. Para ello comprobaremos que los tiempos de ejecución mejoran cuantos más nodos se utilizan para la ejecución, y que la salida esperada sigue siendo la misma.

- Probar diversos ficheros en el *MapReduce*. El objetivo final es verificar la reducción del tiempo de procesamiento usando una ejecución secuencial y con 1, 2 y 4 nodos de un clúster de Hadoop. Las pruebas determinarán la mejora de procesamiento cuanto con respecto al número de nodos.

5.2 Batería de Pruebas

CONFIGURACION EXPERIMENTAL

Para comenzar hemos seleccionado 6 ficheros que tienen un cierto número de transacciones pasando desde las 4100 hasta las 167000. Cada fila obtenida del archivo "csv" contiene datos de todos los equipos de baloncesto que han pasado por la NBA, las columnas son del tipo: año de la liga, id del equipo,…etc. [1].El tamaño de las columnas variara en función del fichero.

Con estos ficheros lo que queremos comprobar es el tiempo que tarda nuestro proceso MapReduce en procesarlos secuencialmente con un nodo y en paralelo con dos y 4 nodos. Hemos asignado 1Gb de memoria para el JobTracker y 1Gb para el Datanode.

En la Tabla 3. Resultado de las Pruebas podemos ver los resultados obtenidos. Para obtener estos datos hemos usado un soporte de un 10% y una confianza de un 20%.

El cálculo del tiempo se ha realizado ejecutando el algoritmo N veces, en nuestro caso N ha sido 10, y calculando la media de los tiempos de las N ejecuciones.

Campos	Transacciones	Secuencial	1 Nodo	2 Nodos	4 Nodos
9	4100	0,140	0,140	0,700	0,340
5	11500	2,678	2,560	1,270	0,640
5	24000	0,170	0,169	0,876	0,480
30	42500	195,780	194,540	89,170	44,164
24	98000	56,370	55,920	27,120	13,770
13	167000	15,926	15,272	8,256	3,987

Tabla 3. Resultado de las Pruebas

Analizando la tabla podemos hacer varias observaciones. En primer lugar, se puede observar que los tiempos de procesamiento secuencial y con un solo nodo son muy similares puesto que es la misma forma de procesamiento, ya que con un nodo no hay posibilidad de paralelización. Observamos también que cuando ejecutamos el programa con dos nodos el tiempo se reduce, esto es debido a que con dos nodos sí que hay ya posibilidad de paralelización por lo que como podemos observar con los datos obtenidos el tiempo de procesamiento cae en un 50% con respecto al procesamiento con un nodo. Esto tiene sentido puesto que con dos nodos realizaremos el trabajo el doble de rápido. Sí incrementamos los nodos, cada nodo hará la parte proporcional del trabajo que le sea asignada en función del número de nodos presentes, por lo cual el tiempo de ejecución se reducirá de forma proporcional al número de nodos.

De forma similar, al pasar de dos a cuatro nodos el tiempo se reduce aproximadamente en un 50%.

En la Tabla 4. `Tiempos con 167000 transacciones` podemos observar de forma gráfica lo que se ha explicado anteriormente en relación a los tiempos de procesamiento. Los datos de la tabla hacen referencia a las estadísticas de un jugador de la NBA: peso, altura, equipo, edad… etc.

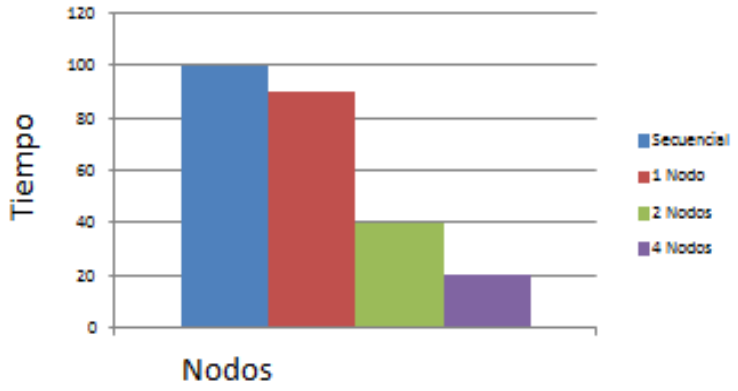


Tabla 4. Tiempos con 167000 transacciones

Hemos realizado una segunda batería de pruebas utilizando un fichero "csv", en el que las el número de transacciones se va duplicando, y el número de columnas se va incrementando de 5 en 5.

Hemos realizado las pruebas ejecutando nuestro algoritmo de modo secuencial, y en paralelo con uno, dos y cuatro nodos. Para obtener estos datos, como en el ejemplo anterior, hemos usado un soporte de un 10% y una confianza de un 20%.

En las tablas 5-8 se muestran los resultados obtenidos para una ejecución secuencial, con un nodo, dos nodos y cuatro nodos, respectivamente. Los tiempos obtenidos están medidos en segundos.

Transacciones/ Campos	5	10	15	20	25	30
20000	0,250	2,270	4,870	9,380	18,540	35,320
40000	0,270	2,300	5,100	9,410	18,600	35,400
60000	0,540	4,600	10,200	18,820	37,200	70,800
80000	1,080	9,200	20,400	37,640	74,400	141,600
100000	2,160	18,400	40,800	75,280	148,800	283,200

Tabla 5. Ejecución Secuencial

Transacciones/ Campos	5	10	15	20	25	30
20000	0,230	2,200	4,870	9,380	18,540	34,320
40000	0,240	2,300	4,700	9,410	18,600	34,410
60000	0,480	4,600	9,400	18,820	37,200	68,820
80000	0,960	9,200	18,800	37,640	74,400	137,640
100000	1,920	18,400	37,600	75,280	148,800	275,280

Tabla 6. Ejecución con un nodo

Transacciones/ Campos	5	10	15	20	25	30
20000	0,110	1,100	2,430	4,680	9,240	17,160
40000	0,118	1,230	2,500	4,700	9,300	17,200
60000	0,236	2,460	5,000	9,400	18,600	34,400
80000	0,472	4,920	10,000	18,800	37,200	68,800
100000	0,944	9,840	20,000	37,600	74,400	137,600

Tabla 7. Ejecución con dos nodos

Transacciones/ Campos	5	10	15	20	25	30
20000	0,050	0,600	1,130	2,250	4,570	8,260
40000	0,054	0,460	1,150	2,310	4,630	8,300
60000	0,108	0,920	2,300	4,620	9,260	16,600
80000	0,216	1,840	4,600	9,240	18,520	33,200
100000	0,432	3,680	9,200	18,480	37,040	66,400

Tabla 8. Ejecución con cuatro nodos

Con esta batería de pruebas llegamos a la misma conclusión que con las pruebas anteriores. El procesamiento de forma secuencial y con un nodo es muy parecido, ya que la forma de ejecutarlo no cambia al no haber posibilidad de paralelización.

En cambio, cuando pasamos a dos nodos vemos una reducción sustancial del tiempo de procesamiento: este cae a la mitad con respecto a la ejecución de forma secuencial. En el caso del uso de cuatro nodos el tiempo de ejecución cae a la mitad con respecto al de dos nodos. Esto es lógico y muy parecido a lo obtenido en la prueba anterior.

CAPITULO 6
CONCLUSIONES Y TRABAJO FUTURO

En este capítulo se detallaran las conclusiones obtenidas al desarrollar el algoritmo "A Priori" e implementarlo con el entorno Hadoop MapReduce.

De igual manera se exponen las oportunidades que han quedado abiertas en el presente documento, para poder seguir avanzando de cara a ampliar nuestros conocimientos tanto de los entornos Hadoop como del concepto Big Data.

6.1 Conclusiones

El trabajo presentando en este documento es un análisis de la implementación del algoritmo "A Priori" en el entorno Hadoop MapReduce.

Este análisis ha sido realizado partiendo de unos conocimientos teóricos tanto sobre el algoritmo "A Priori" como sobre el entorno MapReduce, explicados de una manera sencilla e intuitiva. Los objetivos planteados consistían, en primer lugar, en implementar el algoritmo "A Priori" de una forma secuencial realizando con él una serie de pruebas para comprobar su rendimiento. Posteriormente al algoritmo creado se le aplicó MapReduce para estudiar su comportamiento mediante el procesamiento en paralelo ofrecido por el entorno Hadoop MapReduce. Los resultados obtenidos de estas experiencias han permitido analizar las diferencias más significativas entre los dos modelos de procesamiento.

A continuación se exponen las conclusiones obtenidas una vez realizadas las pruebas sobre el algoritmo en sus dos versiones.

El algoritmo "A priori" es uno de los algoritmos más potentes y fiables para sacar patrones en grandes conjuntos de datos, puesto que al generar primero los candidatos frecuentes se reduce el espacio de búsqueda y se aumenta la eficiencia de la misma.

El entorno MapReduce es el mejor sistema a la hora de paralelizar el procesamiento de un algoritmo, debido a que toda la labor de distribución del trabajo la realiza el entorno Hadoop de manera interna, y el usuario solo debe de encargarse de crear las dos funciones necesarias para el procesamiento de los datos.

El procesamiento de un algoritmo de forma secuencial es mucho más ineficiente que el procesamiento de forma paralela, en nuestro caso con el entorno MapReduce. Esto se debe a que de forma secuencial la misma máquina ha de realizar todo el trabajo de procesamiento, y sin embargo en el entorno Hadoop, el trabajo se distribuye entre N máquinas, (dependiendo del número de nodos que queramos usar) porque el trabajo se agiliza de forma considerable como ha sido demostrado en las pruebas.

6.2 Trabajo Futuro

El trabajo futuro que sería necesario realizar para seguir avanzando en la aplicación del análisis asociativo a grandes conjuntos de datos puede resumirse en las siguientes líneas:

- Realizar las pruebas de rendimiento cambiando los valores de Confianza y Soporte Mínimo para ver el comportamiento del algoritmo.

- Someter al algoritmo a más pruebas con ficheros de millones de datos, e incrementar el número de nodos utilizados en la ejecución del algoritmo.

- Realizar una nueva implementación del algoritmo "A Priori", mejorando el rendimiento de forma secuencial.

- Realizar una nueva implementación de las funciones del entorno MapReduce para mejorar el rendimiento al paralelizar el algoritmo "A Priori".

REFERENCIAS

[1] Tan P, Steinbach M, Kumar V.2014. Introduction to Data Mining.Pearson.1ª Edicion.732pp.

[2] Berzal F.2015. Reglas de asociación. Departamento de Ciencias de la Computación e IA, Universidad de Granada.

[3] Gironés Roig J. 2013. Algoritmos. Universitat Omertat de Catalunya.48-52.

[4] Saed Sayad http://www.saedsayad.com/association_rules.htm. 2015.

[5] Perera **S**, Gunarathne T. 2013. Hadoop Mapreduce Cookbook EBook Edition.PACKT.2ªEdición.300pp

[6] Vlad Korolev http://ebiquity.umbc.edu/Tutorials/Hadoop/00%20-%20Intro.html. Junio de 2015

[7] Luis Miguel Gracia https://unpocodejava.wordpress.com/2013/05/02/mapreduce-con-mongodb/. Junio de 2015.

[8] Rokach L, Maimon O.2008. Data mining with decision trees: theory and applications. World Scientific.Pearson.445pp.

[9] Wikipedia. https://es.wikipedia.org/wiki/Tabla_hash.Junio 2015.

[10]Natan Magnus http://www.codemiles.com/java/apriori-algorithm-java-code-t5700.html. Enero 2015.

[11] Google Code https://code.google.com/p/coms6111-proj3/source/browse/trunk/src/apriori/Apriori.java?spec=svn7&r=7.Enero 2015.

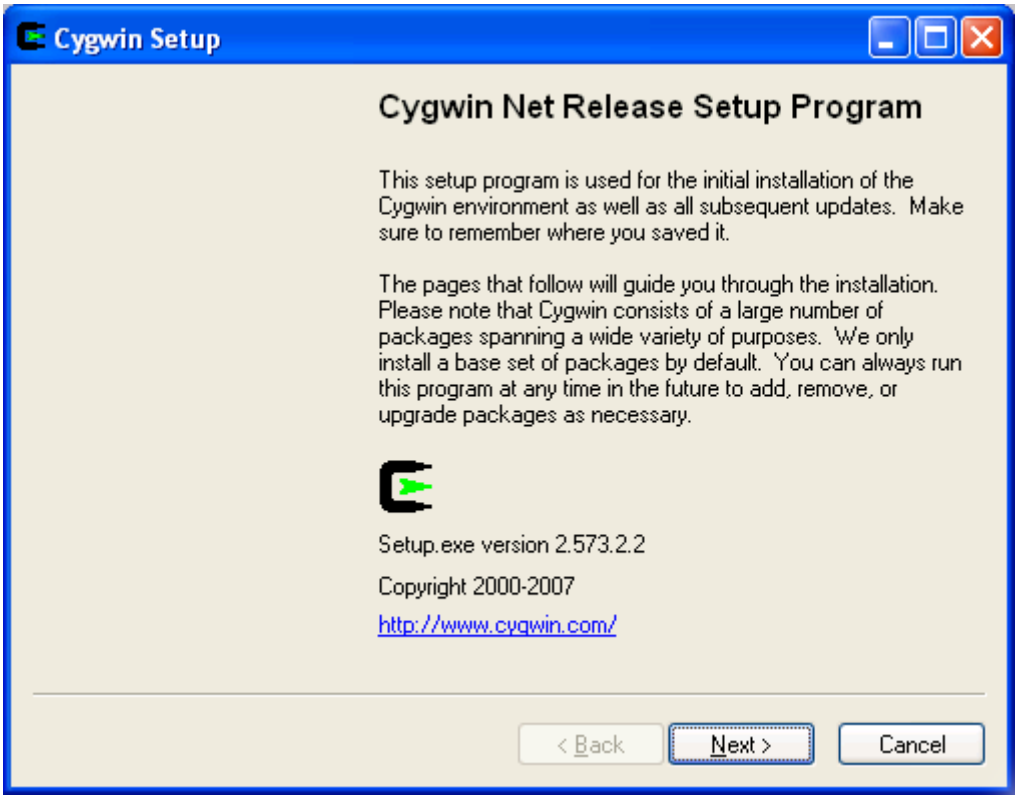
ANEXOS

Anexo A: Instalación de clúster en Cygwin y Hadoop.

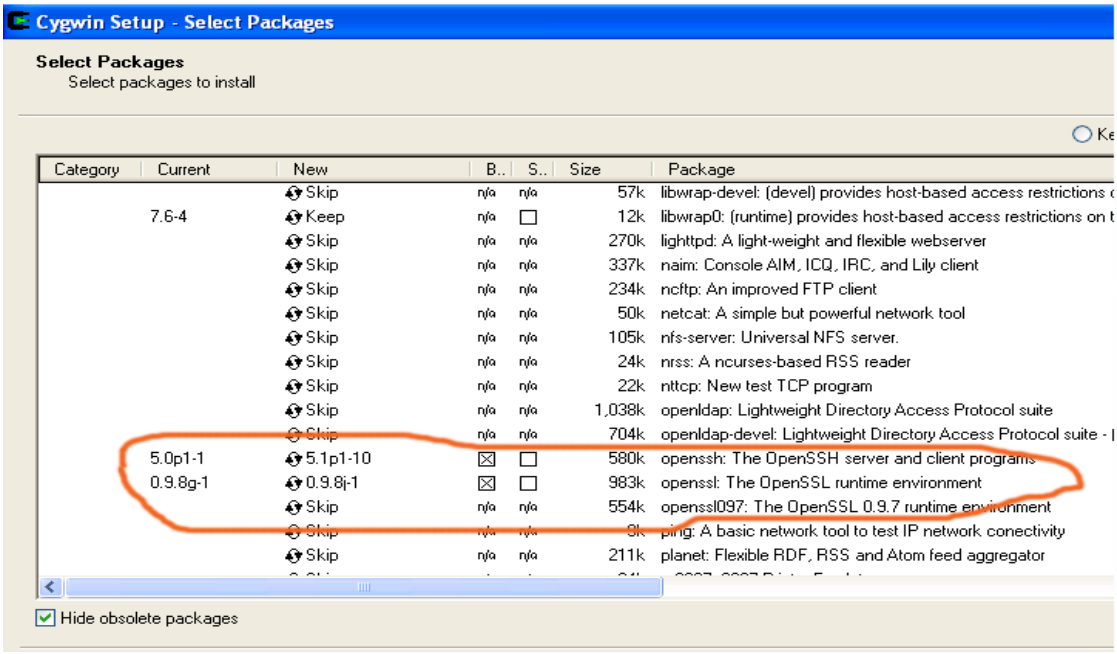
A.1 Instalar Cygwin.

Primero deberemos descargarnos cygwin desde su página de internet que es la siguiente: https://www.cygwin.com/

Una vez descargado procederemos a su instalación pulsando sobre el ejecutable descargado, y nos aparecerá la siguiente pestaña.

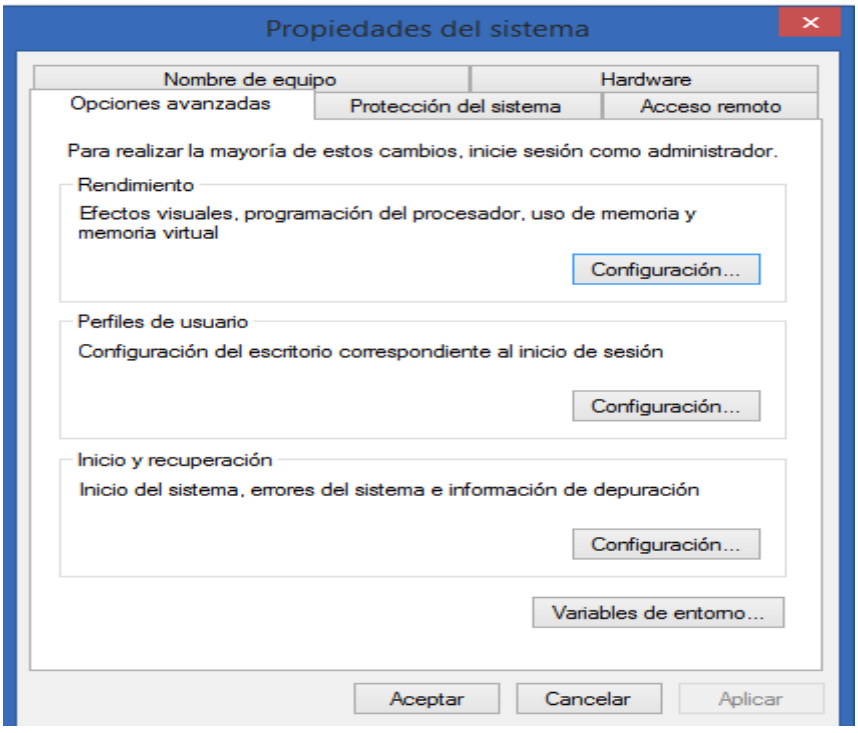


Pulsamos next hasta que veamos la pantalla de selección de paquetes, hay que asegurarse de que tenemos seleccionado el paquete openssh que es necesario para el correcto funcionamiento del plugin de eclipse y el clúster de hadoop.

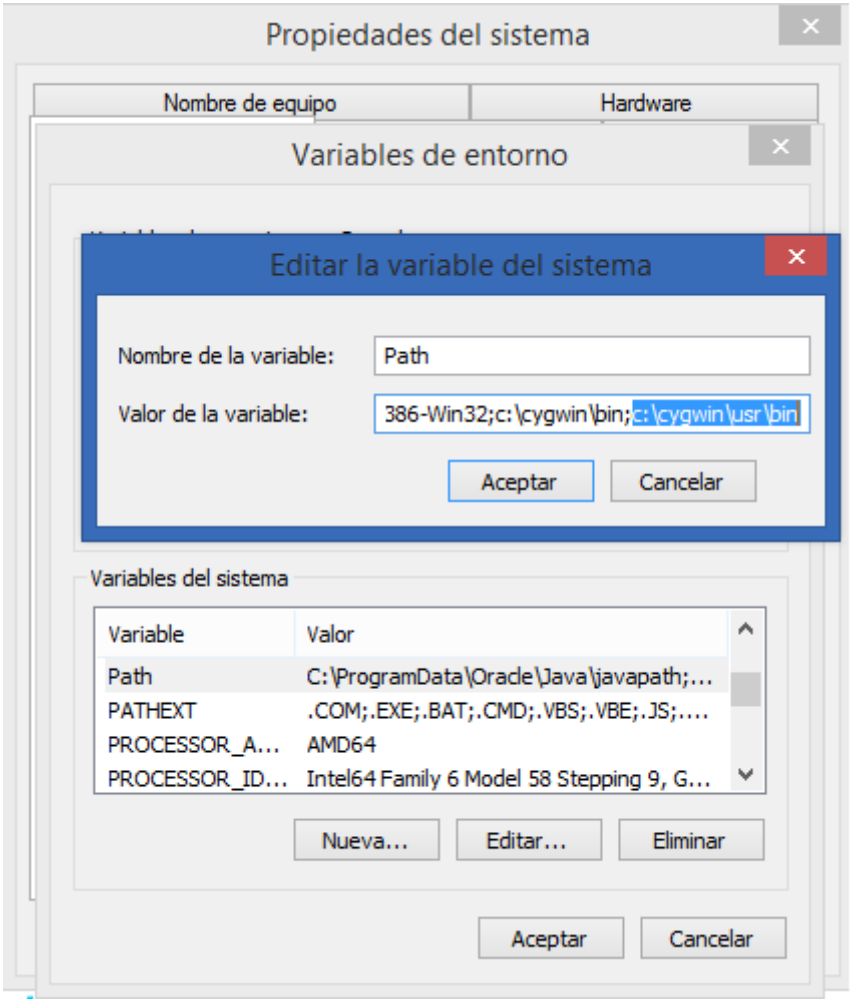


Una vez seleccionados los paquetes pulsamos el botón next para acabar con la instalación.

El siguiente paso es configurar las variables de entorno, para que eclipse pueda acceder a los comandos de cygwin. Para acceder a las variables de entorno debemos de seguir los siguientes pasos, primero deberemos ir a Mi PC y pulsar doble clic en el. Una vez pulsado deberemos pulsar propiedades y nos saldrá la siguiente ventana



En la ventana anterior pulsamos en variables de entorno y vamos a la que pone path y pulsamos editar.



Y al final de la variable de entorno añadimos lo siguiente:
C:\cygwin\bin;c:\cygwin\usr\bin
Una vez añadido esto a las variables de entorno cerramos todas las variables de entorno y pasamos a la siguiente parte que será configurar el cluster

A.2 Configurar el Demonio ssh.

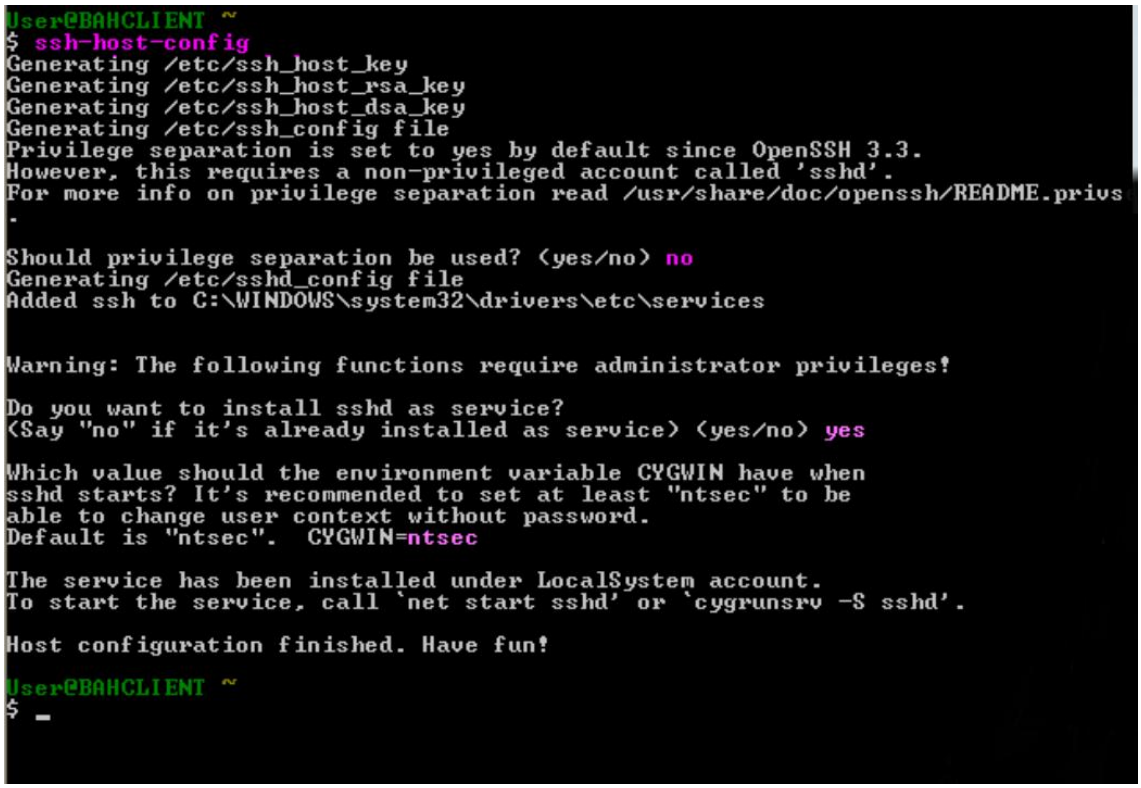
Para comenzar esta parte abrimos la terminar de Cygwin y ejecutamos el siguiente comando: *ssh-host-config.*

A continuación nos saldrán una serie de preguntas en la terminal a las que deberemos contestar con lo siguiente:

A la pregunta "*privelege separation should be used"* deberemos responder no.

Cuando pregunte "*sshd should be installed as a service"* deberemos de responder si.
Y por último cuando pregunte acerca de la variable de entorno Cygwin deberemos de poner ntsec.

A continuación ponemos un ejemplo de cómo va a ser lo que nos salga en la consola ha de tenerse en cuenta de que ese resultado puede variar.



.

Anexo B: Código del algoritmo a priori usando HashSet.

En esta parte de la memoria explicaremos el código de algoritmo a priori paso por paso como lo hemos ido realizando.

```
/**
 * Does the Apriori Algorithm
 * @param minSupport
 * @param minConfidence
 * @throws IOException
*/
public void doApriori(Double minSupport, Double minConfidence, String
filename) throws IOException {
       fileMap = FileHelper.parseFile(filename);
       currentItemSet = DataHelper.getLOneSet();
       largeItemSet.addAll(currentItemSet);
             while(currentItemSet.size() > 0) {
                currentItemSet = getNextLevelItemSet(currentItemSet);
                }
                      System.out.println("==Large itemsets (
min_support="+ minSupport*100 + "% )");
                      printLargeItemSet();
                      System.out.println();
                      System.out.println("==High-confidence
association rules"+ "( min_conf="+minConfidence*100 + "% )");
                      generateAssociationRules(minConfidence);
              }
```

Esta es el algoritmo principal del código como podemos ver se le pasan como parámetros en soporte y la confianza mínimos para las reglas y el nombre del fichero.
Lo primero que hace este método es llamar al método parse file que está declarado en la clase FileHelper que podemos ver a continuación.
Este método se encarga de ir leyendo todas transacciones del fichero (líneas del fichero) e ir metiéndolas en `Map<Integer, Set<String>> fileMap`.

Este map contendrá como clave la línea de fichero y como valor el contenido de la línea del fichero.
Aparte de realizar esto el método se encarga de llamar al método `updateFrequency`.
Este método se encarga de meter en un map las líneas de fichero con clave el número de veces que aparecen y como valor el contenido de la línea del fichero.
Por último se encarga de llamar al método `setNumTransactions` que se encarga de guardar el número de líneas del fichero.

```
public class FileHelper {
 /**
    * Parses the CSV file and fills up the fileMap
    * @param filename
    * @return
    * @throws IOException
  */
 public static Map<Integer, Set<String>> parseFile(String filename)
throws IOException {
   Map<Integer, Set<String>> fileMap = new HashMap<Integer,
   Set<String>>();
   Set<String> lineSet;
   String input ="";
   BufferedReader reader = new BufferedReader(new
FileReader(filename));
   Integer lineCount = 0;
     while((input = reader.readLine()) != null) {
              ++lineCount;
               lineSet = new HashSet<String>();
               String temp [] = input.split(",");
                 int i = 0;

                    /**
                     * Adds to the lineSet and also populates
                     * the frequencyMap using the DataHelper
                     */
                    while (i < temp.length) {

                           lineSet.add(temp[i]);
                           DataHelper.updateFrequency(temp[i]);
                           i++;

                       }
                       fileMap.put(lineCount, lineSet);
               }

               DataHelper.setNumTransactions(lineCount);
               return fileMap;
        }
}
```

El siguiente paso del código es obtener los "*currentItemSet*" es decir viendo en *filemap* anterior obtenido al llamar al método *updateFrequency*
Los valores que cumplen que la clave del map/numero de transacciones sean mayor que el soporte mínimo.
A continuación adjuntamos el código de la función *getLoneSet* para poder ver más claramente lo anteriormente explicado.

```
package helper;

import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;
import valueObjects.LargeItemSetVO;


 /**
   * Get the L1 items
   * @return
   */
public static Set<Set<String>> getLOneSet() {
                Set<Set<String>> LOneSet = new HashSet<Set<String>>();
   Set<String> item = null;
   for (String s: frequencyMap.keySet()) {
      Double currSuport = frequencyMap.get(s).doubleValue() /
numberOfTransactions.doubleValue();
             if (currSuport >= minSupport) {
                    item = new HashSet<String>();
                    item.add(s);
                    LOneSet.add(item);
                   largeItemSetWithSupport.add(new
LargeItemSetVO(item, currSuport));
                        }
               }
            return LOneSet;
        }
```

Como podemos ver aparte de lo anteriormente explicado este método hace dos cosas más se encarga de añadir los valores encontrados a un `Set<Set<String>> LOneSet`. Esto es para que se ordenen internamente las líneas y luego lo añade a `largeItemSetWithSupport` que se encarga de guardar el valor con su support actual para usarlo para generar los candidatos frecuentes.
A continuación vamos a generar los frecuentes itemset para ello usaremos la siguiente parte del algoritmo a priori y las funciones que destacamos a continuación.

```
while(currentItemSet.size() > 0) {
                currentItemSet = getNextLevelItemSet(currentItemSet);
                }
```

Como vemos recorre la lista de itemset que tenemos y llama a la función que es mostrada más abajo la cual generara los frecuent itemsets.

```
*/


public Set<Set<String>> getNextLevelItemSet(Set<Set<String>>
curItemSet){

 Set<Set<String>> cloneSet = new HashSet<Set<String>>(curItemSet);
 Set<Set<String>> nextLevel = new HashSet<Set<String>>();

      for(Set<String> set : curItemSet) {
          for(Set<String> set1 : cloneSet){
```

```
                    if (!set.equals(set1)) {
                        Set<String> toAdd = new HashSet<String>(set);
                                    toAdd.addAll(set1);
                                     nextLevel.add(toAdd);
                                       }
                                }
                            }
          nextLevel = DataHelper.getFrequentItems(nextLevel, fileMap);
          largeItemSet.addAll(nextLevel);
          return nextLevel;
              }
}
```

Esta función se encarga de generar los candidatos frecuentes, para ello clona los ItemSets en un *HashSet* y va recorriendo cada transacción del *HashSet* de *ItemSets* frecuentes e introduciéndolos en un *HashSet* que contendrá todos los elementos siguientes del *Hashmap* para hacer la comparación. Con este *HasSet* de siguiente niveles y todas las transacciones que hemos leído del fichero vamos a la generación de los *frequentItemsets* llamando a las siguientes funciones.

```
public static Set<Set<String>> getFrequentItems(Set<Set<String>>
allCurrentItemSets, Map<Integer, Set<String>> fileMap) {

Set<Set<String>> frequentSets = new HashSet<Set<String>>();

for (Set<String> set: allCurrentItemSets) {

     if (checkSetIsFrequent(fileMap, set)) {
                  frequentSets.add(set);
                        }
               }
          return frequentSets;
        }
```

```
/**
          * Checks if it is frequent and puts it onto the
          * LargeItemSetWithSupport Set.
          * @param fileMap
          * @param set
          * @return
          */

 private static boolean checkSetIsFrequent(Map<Integer, Set<String>>
fileMap, Set<String> set) {

Integer count = 0;

for (Integer key: fileMap.keySet()) {
```

```
Set<String> row = fileMap.get(key);

     if (row.containsAll(set))
                                  ++count;
                }
     if ( ( count.doubleValue() / numberOfTransactions.doubleValue() )
>= minSupport) {
     largeItemSetWithSupport.add(new LargeItemSetVO(set,
count.doubleValue() / numberOfTransactions.doubleValue()));

      return true;
                }
                return false;
        }
```

La función de generar los *frecuentItemSets* recibe como parámetros lo explicado anteriormente Set con los *nextLevels* y el *frecuencyMap* generado parseando transacciones del sistema para ello recorremos todo el Set de *nextlevel* y lo pasamos a la función de *chekIsFrecuent* en caso de que se cumpla se añade a un Set que es lo que devolverá la función.

La función que comprueba si el elemento es frecuente funciona de la siguiente manera recorre el *fileMap* y comprueba que contenga el elemento que le hemos pasado en el Set de *nextLeveles* en el caso de que exista aumenta el count, para comprobar si es frecuente realiza el siguiente algoritmo `( count.doubleValue() / numberOfTransactions.doubleValue() ) >= minSupport)` .

En el caso de que se cumpla devuelve true y con ello habremos generado todos los *frequent itemSets*, los cuales los necesitaremos para la siguiente parte del algoritmo que es generar las reglas de asociación.
A continuación mostraremos el método que se encarga de generar las reglas de asociación.

```
/**

      * private method to generate association rules

*/

private void generateAssociationRules(Double minConfidence) {
       ArrayList<AssociationObject> associations = new
ArrayList<AssociationObject>();
       ArrayList<LargeItemSetVO> set = new
ArrayList<LargeItemSetVO>(DataHelper.getLargeItemSetWithSupport());
       for(LargeItemSetVO vo: set) {
                              /* Get one item on the rhs */
            if (vo.getItems().size() == 1) {
                    String item = vo.getItems().iterator().next();
            for (LargeItemSetVO other : set) {
                if (other.getItems().contains(item) &&
other.getItems().size() > 1) {

/*Propose an association if > than confidence*/

         Set<String> toOutput = new HashSet<String>(other.getItems());

            toOutput.remove(item);
```

```
                                                        if
((other.getSupport().doubleValue() /
DataHelper.getSupport(toOutput).doubleValue()) >= minConfidence) {

associations.add(new AssociationObject(toOutput, vo.getItems(),
(other.getSupport().doubleValue() /
DataHelper.getSupport(toOutput).doubleValue()), other.getSupport()));
                                                        }
                                                    }
                                                }
                                            }
                                        }
            Collections.sort(associations);
            printAssociations(associations);
        }
```

Anexo C: Código del algoritmo a priori usando HashTree.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

//
//  Author: Su Yibin, under the supervision of Howard Hamilton and
//          Mengchi Liu
//  Copyright: University of Regina and Su Yibin, April 2000
//  No reproduction in whole or part without maintaining this
copyright
//  notice and imposing this condition on any subsequent users.
//

//---- apriori.java

//---- input file need:
//----   1. config.txt
//----      four lines, each line a integer
//----      item number, transaction number , minsup
//----   2. transa.txt

import java.io.*;
import java.util.*;

//-----------------------------------------------------------------
//  Class Name : apriori
//  Purpose    : main program class
//-----------------------------------------------------------------
public class AlgorimoAprioriHashTree
{
  public static void main(String[] args) throws IOException
  {
    aprioriProcess process1=new aprioriProcess();
    System.exit(0);
  }
}

//-----------------------------------------------------------------
```

```
//  Class Name : aprioriProcess
//  Purpose    : main processing class
//-----------------------------------------------------------
class aprioriProcess
{
  private final int HT=1; // state of tree node (hash table or
  private final int IL=2; // itemset list)
  int N; // total number of items per transaction
  int M; // total number of transactions
  Vector<Vector<String>> largeitemset=new Vector<Vector<String>>();
  Vector<candidateelement> candidate=new Vector<candidateelement>();
  int minsup; // minimum support to make frequent
  String fullitemset;
  String configfile="config.txt"; // default configuration file
  String transafile="prueba2.txt"; // default transaction file

//-----------------------------------------------------------
//  Class Name : candidateelement
//  Purpose    : object that will be stored in Vector candidate
//             : include 2 item
//             : a hash tree and a candidate list
//-----------------------------------------------------------
  class candidateelement
  {
    hashtreenode htroot;
    Vector candlist;
  }

//-----------------------------------------------------------
//  Class Name : hashtreenode
//  Purpose    : node of hash tree
//-----------------------------------------------------------
  class hashtreenode
  {
    int nodeattr; //  IL or HT
    int depth; // the current itemset depth (ie: 1-itemset, 2-
itemsets, etc)
    Hashtable<String, hashtreenode> ht;
    Vector<itemsetnode> itemsetlist;

    public void hashtreenode()
    {
      nodeattr=HT;
      ht=new Hashtable<String, hashtreenode>();
      itemsetlist=new Vector<itemsetnode>();
      depth=0;
    }

    public void hashtreenode(int i)
    {
      nodeattr=i;
      ht=new Hashtable<String, hashtreenode>();
      itemsetlist=new Vector<itemsetnode>();
      depth=0;
    }
  }

//-----------------------------------------------------------
//  Class Name : itemsetnode
//  Purpose    : node of itemset
//-----------------------------------------------------------
```

```
  class itemsetnode
  {
    String itemset;
    int counter;

    public itemsetnode(String s1,int i1)
    {
      itemset=new String(s1);
      counter=i1;
    }

    public itemsetnode()
    {
      itemset=new String();
      counter=0;
    }

    public String toString()
    {
      String tmp=new String();
      tmp=tmp.concat("<\"");
      tmp=tmp.concat(itemset);
      tmp=tmp.concat("\",");
      tmp=tmp.concat(Integer.toString(counter));
      tmp=tmp.concat(">");
      return tmp;
    }
  }

//-------------------------------------------------------------
//  Method Name: getconfig
//  Purpose    : open file config.txt
//             : get the total number of items of transaction file
//             : and the total number of transactions
//             : and minsup
//-------------------------------------------------------------
  public void getconfig() throws IOException
  {
    FileInputStream file_in;
    BufferedReader data_in;
    String oneline=new String();
    //open the config file and load the values
    try
    {
      file_in = new FileInputStream(configfile);
      data_in = new BufferedReader(new InputStreamReader(file_in));

      //number of transactions
      oneline=data_in.readLine();
      N=Integer.valueOf(oneline).intValue();

      //number of items
      oneline=data_in.readLine();
      M=Integer.valueOf(oneline).intValue();

      //minsup
      oneline=data_in.readLine();
      minsup=Integer.valueOf(oneline).intValue();

      //output config info to the user
```

```
      System.out.print("\nInput configuration: "+N+" items, "+M+"
transactions, ");
      System.out.println("minsup = "+minsup+"%");
      System.out.println();
    }
    catch (IOException e)
    {
      System.out.println(e);
    }
  }

//-------------------------------------------------------------
//  Method Name: getitemat
//  Purpose    : get an item from an itemset
//             : get the total number of items of transaction file
//  Parameter  : int i : i-th item ; itemset : string itemset
//  Return     : String : the item at i-th in the itemset
//-------------------------------------------------------------
  public String getitemat(int i,String itemset)
  {
    String str1=new String(itemset); //copy the itemset into a new
string
    StringTokenizer st=new StringTokenizer(itemset); //create a
tokenizer
    int j;

    //if trying to get an position outside of the set, notify user of
error
    if (i > st.countTokens())
      System.out.println("eRRor! in getitemat, !!!!");

    //loop through until get the token in the ith position
    for (j=1;j<=i;j++)
      str1=st.nextToken();

    //return the integer value of the ith position
    return(str1);
  }

//-------------------------------------------------------------
//  Method Name: itesetsize
//  Purpose    : get item number of an itemset
//  Parameter  : itemset : string itemset
//  Return     : int : the number of item of the itemset
//-------------------------------------------------------------
  public int itemsetsize(String itemset)
  {
    StringTokenizer st=new StringTokenizer(itemset);//tokenize the
itemset
    return st.countTokens();//number of tokens is the size of the set
  }

//-------------------------------------------------------------
//  Method Name: gensubset
//  Purpose    : generate all subset given an itemset
//  Parameter  : itemset
//  Return     : a string contains all subset deliminated by ","
//             : e.g. "1 2,1 3,2 3" is subset of "1 2 3"
//-------------------------------------------------------------
  public String gensubset(String itemset)
  {
```

```
    int len=itemsetsize(itemset);
    int i,j;
    String str1;
    String str2=new String();
    String str3=new String();

    if (len==1)
      return null;
    for (i=1;i<=len;i++)
    {
      StringTokenizer st=new StringTokenizer(itemset); //tokenize the
itemset
      str1=new String();
      //add each token to str1 and put the spaces in
      for (j=1;j<i;j++)
      {
        str1=str1.concat(st.nextToken());
        str1=str1.concat(" ");
      }
      //
      str2=st.nextToken();
      for (j=i+1;j<=len;j++)
      {
        str1=str1.concat(st.nextToken());
        str1=str1.concat(" ");
      }

      if (i!=1)
        str3=str3.concat(",");

      str3=str3.concat(str1.trim());
    }
    return str3;

  }

//-----------------------------------------------------------------
//  Method Name: createcandidate
//  Purpose    : generate candidate n-itemset
//  Parameter  : int n : n-itemset
//  Return     : Vector : candidate is stored in a Vector
//-----------------------------------------------------------------
  public Vector createcandidate(int n)
  {
    Vector<String> tempcandlist=new Vector<String>();
    Vector ln_1=new Vector();
    int i,j,length1;
    String cand1=new String();
    String cand2=new String();
    String newcand=new String();

//    System.out.println("Generating "+n+"-candidate item set ....");
    //if its the 1-itemset, just add all the items to the list
    if (n==1)
      for (i=1;i<=N;i++)
        tempcandlist.addElement(Integer.toString(i));
    //if its 2 or more itemset
    else
    {
      ln_1=(Vector)largeitemset.elementAt(n-2);
      length1=ln_1.size();
```

```
      //for each item in the set
      for (i=0;i<length1;i++)
      {
        cand1=(String)ln_1.elementAt(i);
        //check from the next one until the end and make new item sets
        for (j=i+1;j<length1;j++)
        {
          cand2=(String)ln_1.elementAt(j);
          newcand=new String();
          if (n==2) //if depth = 2, then no formula to determine which
ones can combine
          {
            newcand=cand1.concat(" ");
            newcand=newcand.concat(cand2);
            tempcandlist.addElement(newcand.trim());
          }
          else //first n-2 items in the itemset must be same for
itemsets to be combined
          {
            int c;
            String i1,i2;
            boolean same=true;

            for (c=1;c<=n-2;c++)
            {
              i1=getitemat(c,cand1);
              i2=getitemat(c,cand2);
              if ( i1.compareToIgnoreCase(i2)!=0 )
              {
                same=false;
                break;
              }
              else
              {
                newcand=newcand.concat(" ");
                newcand=newcand.concat(i1);
              }
            }
            if (same) //if the first n-2 items are the same, combine
the sets
            {
              i1=getitemat(n-1,cand1);
              i2=getitemat(n-1,cand2);
              newcand=newcand.concat(" ");
              newcand=newcand.concat(i1);
              newcand=newcand.concat(" ");
              newcand=newcand.concat(i2);
              tempcandlist.addElement(newcand.trim());
            }
          } //end if n==2 else
        } //end for j
      } //end for i
    } //end if n==1 else

    if (n<=2)
      return tempcandlist;

    Vector<String> newcandlist=new Vector<String>();
    //for each candidate, if already has the itemset (tokenizer splits
at ",") then don't add it
    for (int c=0; c<tempcandlist.size(); c++)
```

```
    {
      String c1=(String)tempcandlist.elementAt(c);
      String subset=gensubset(c1);
      StringTokenizer stsubset=new StringTokenizer(subset,",");
      boolean fake=false;
      while (stsubset.hasMoreTokens())
        if (!ln_1.contains(stsubset.nextToken()))
        {
          fake=true;
          break;
        }
      if (!fake)
        newcandlist.addElement(c1);
    }

    return newcandlist;

  }

//-------------------------------------------------------------
//  Method Name: createcandidatehashtre
//  Purpose    : generate candidate hash tree
//  Parameter  : int n : n-itemset
//  Return     : hashtreenode : root of the hashtree
//-------------------------------------------------------------
  public hashtreenode createcandidatehashtree(int n)
  {
    int i,len1;
    hashtreenode htn=new hashtreenode();


    if (n==1)
      htn.nodeattr=IL;
    else
      htn.nodeattr=HT;

    len1=((candidateelement)candidate.elementAt(n-1)).candlist.size();
    for (i=1;i<=len1;i++)
    {
      String cand1=new String();
      cand1=(String)((candidateelement)candidate.elementAt(n-
1)).candlist.elementAt(i-1);
      genhash(1,htn,cand1);
    }

    return htn;

  }


//-------------------------------------------------------------
//  Method Name: genhash
//  Purpose    : called by createcandidatehashtree
//             : recursively generate hash tree node
//  Parameter  : htnf is a hashtreenode (when other method call this
method,it is the root)
//             : cand : candidate itemset string
//             : int i : recursive depth,from i-th item, recursive
//  Return     :
//-------------------------------------------------------------
  public void genhash(int i, hashtreenode htnf, String cand) {
```

```
    int n=itemsetsize(cand);
    if (i==n) {
      htnf.nodeattr=IL;
      htnf.depth=n;
      itemsetnode isn=new itemsetnode(cand,0);
      if (htnf.itemsetlist==null)
        htnf.itemsetlist=new Vector<itemsetnode>();
      htnf.itemsetlist.addElement(isn);
    }
    else {
      if (htnf.ht==null)
        htnf.ht=new Hashtable<String, hashtreenode>(HT);
      if (htnf.ht.containsKey((getitemat(i,cand)))) {
        htnf=(hashtreenode)htnf.ht.get((getitemat(i,cand)));
        genhash(i+1,htnf,cand);
      }
      else {
        hashtreenode htn=new hashtreenode();
        htnf.ht.put((getitemat(i,cand)),htn);
        if (i==n-1) {
          htn.nodeattr=IL;
          //Vector isl=new Vector();
          //htn.itemsetlist=isl;
          genhash(i+1,htn,cand);
        }
        else {
          htn.nodeattr=HT;
          //Hashtable ht=new Hashtable();
          //htn.ht=ht;
          genhash(i+1,htn,cand);
        }
      }
    }
  }

//-------------------------------------------------------------
//  Method Name: createlargeitemset
//  Purpose    : find all itemset which have their counters>=minsup
//  Parameter  : int n : n-itemset
//  Return     :
//-------------------------------------------------------------
  public void createlargeitemset(int n)
  {
    Vector candlist=new Vector();
    Vector<String> lis=new Vector<String>(); //large item set
    hashtreenode htn=new hashtreenode();
    int i;


    candlist=((candidateelement)candidate.elementAt(n-1)).candlist;
    htn=((candidateelement)candidate.elementAt(n-1)).htroot;

    getlargehash(0,htn,fullitemset,lis);

    largeitemset.addElement(lis);

  }


//-------------------------------------------------------------
```

```
//  Method Name: getlargehash
//  Purpose    : recursively traverse candidate hash tree
//             : to find all large itemset
//  Parameter  : htnf is a hashtreenode (when other method call this
method,it is the root)
//             : cand : candidate itemset string
//             : int i : recursive depth
//             : Vector lis : Vector that stores large itemsets
//  Return     :
//-----------------------------------------------------------------
  public void getlargehash(int i,hashtreenode htnf,String
transa,Vector<String> lis)
  {
    Vector tempvec=new Vector();
    int j;

    if (htnf.nodeattr==IL)
    {
      tempvec=htnf.itemsetlist;
      for (j=1;j<=tempvec.size();j++)
        if (((itemsetnode)tempvec.elementAt(j-1)).counter >= ((minsup
* M) / 100))
          lis.addElement( ((itemsetnode)tempvec.elementAt(j-
1)).itemset );
    }
    else
    {
      if (htnf.ht==null)
        return;
      for (int b=i+1;b<=N;b++)
        if (htnf.ht.containsKey((getitemat(b,transa))))

getlargehash(b,(hashtreenode)htnf.ht.get((getitemat(b,transa))),transa
,lis);
    }
  }

//-----------------------------------------------------------------
//  Method Name: transatraverse
//  Purpose    : read each transaction, traverse hashtree,
//               incrment approporiate itemset counter.
//  Parameter  : int n : n-itemset
//  Return     :
//-----------------------------------------------------------------
  public void transatraverse(int n)
  {
    FileInputStream file_in;
    BufferedReader data_in;
    String oneline=new String();
    int i=0,j=0;
    String transa;
    hashtreenode htn=new hashtreenode();
    StringTokenizer st;
    String str0;
    int numRead=0;

    htn=((candidateelement)candidate.elementAt(n-1)).htroot;
    try
    {

      file_in = new FileInputStream(transafile);
```

```
      data_in = new BufferedReader(new InputStreamReader(file_in));

      while ( true )
      {
        transa=new String();
        oneline=data_in.readLine();
        numRead++;

        //if there are no more transactions, break the loop
        if ((oneline==null)||(numRead > M))
          break;

        st=new StringTokenizer(oneline.trim());
        j=0;

        //check each item in the transaction and add to transaction
string
        while ((st.hasMoreTokens()) && j < N)
        {
            j++;
            str0=st.nextToken();
            i=Integer.valueOf(str0).intValue();
            //add to string indicating what items are present in this
transaction
            if (i!=0)
            {
                transa=transa.concat(" ");
                transa=transa.concat(Integer.toString(j));
            }
        }
        transa=transa.trim();
        transatrahash(0,htn,transa);
      }
    }
    catch (IOException e)
    {
      System.out.println(e);
    }
  }

//---------------------------------------------------------------
//  Method Name: transatrahash
//  Purpose    : called by transatraverse
//             : recursively traverse hash tree
//  Parameter  : htnf is a hashtreenode (when other method call this
method,it is the root)
//             : cand : candidate itemset string
//             : int i : recursive depth,from i-th item, recursive
//  Return     :
//---------------------------------------------------------------
  public void transatrahash(int i,hashtreenode htnf,String transa)
  {
    Vector itemsetlist=new Vector();
    int j,lastpos,len;
    String d;
    itemsetnode tmpnode=new itemsetnode();
    StringTokenizer st;

    if (htnf.nodeattr==IL)
    {
      itemsetlist=(Vector)htnf.itemsetlist;
```

```
      len=itemsetlist.size();
      for (j=0;j<len;j++)
      {
        st = new StringTokenizer(transa);
        tmpnode=(itemsetnode)itemsetlist.elementAt(j);
        d=getitemat(htnf.depth,tmpnode.itemset);

        while(st.hasMoreTokens())
        {
            if(st.nextToken().compareToIgnoreCase(d)==0)
                ((itemsetnode)(itemsetlist.elementAt(j))).counter++;
        }

      }
      return;
    }
    else  //HT
      for (int b=i+1;b<=itemsetsize(transa);b++)
        if (htnf.ht.containsKey((getitemat(b,transa))))

transatrahash(i,(hashtreenode)htnf.ht.get((getitemat(b,transa))),trans
a);

  }

//---------------------------------------------------------------
//  Method Name: aprioriProcess()
//  Purpose    : main processing method
//  Parameters :
//  Return     :
//---------------------------------------------------------------
  public aprioriProcess()  throws IOException
  {
    candidateelement cande;
    int k=0;
    Vector large=new Vector();
    Date d=new Date();
    long s1,s2;

    System.out.println();
    System.out.println("Algorithm apriori starting now.....");
    System.out.println();

    getconfig();

    fullitemset=new String();
    fullitemset=fullitemset.concat("1");
    for (int i=2;i<=N;i++)
    {
      fullitemset=fullitemset.concat(" ");
      fullitemset=fullitemset.concat(Integer.toString(i));
    }

    //start time
    d=new Date();
    s1=d.getTime();

    while (true)
    {
      k++;
      cande=new candidateelement();
```

```
       cande.candlist=createcandidate(k);


       if (cande.candlist.isEmpty())
       break;

       cande.htroot=null;
       candidate.addElement(cande);

       ((candidateelement)candidate.elementAt(k-
1)).htroot=createcandidatehashtree(k);


       transatraverse(k);

       createlargeitemset(k);
       System.out.println("Frequent "+k+"-itemsets:");
       System.out.println((Vector)(largeitemset.elementAt(k-1)));
     }

     hashtreenode htn=new hashtreenode();
     htn=((candidateelement)candidate.elementAt(k-2)).htroot;

     //end time
     d=new Date();
     s2=d.getTime();
     System.out.println();
     System.out.println("Execution time is: "+((s2-s1)/(double)1000) +
" seconds.");

   }
```